Operativni sistemi 2

Vežbe 1
Raspoređivanje procesa

Lista predmeta:
IR smer - 13e113os2@lists.etf.rs
SI Smer - 13s113os2@lists.etf.rs

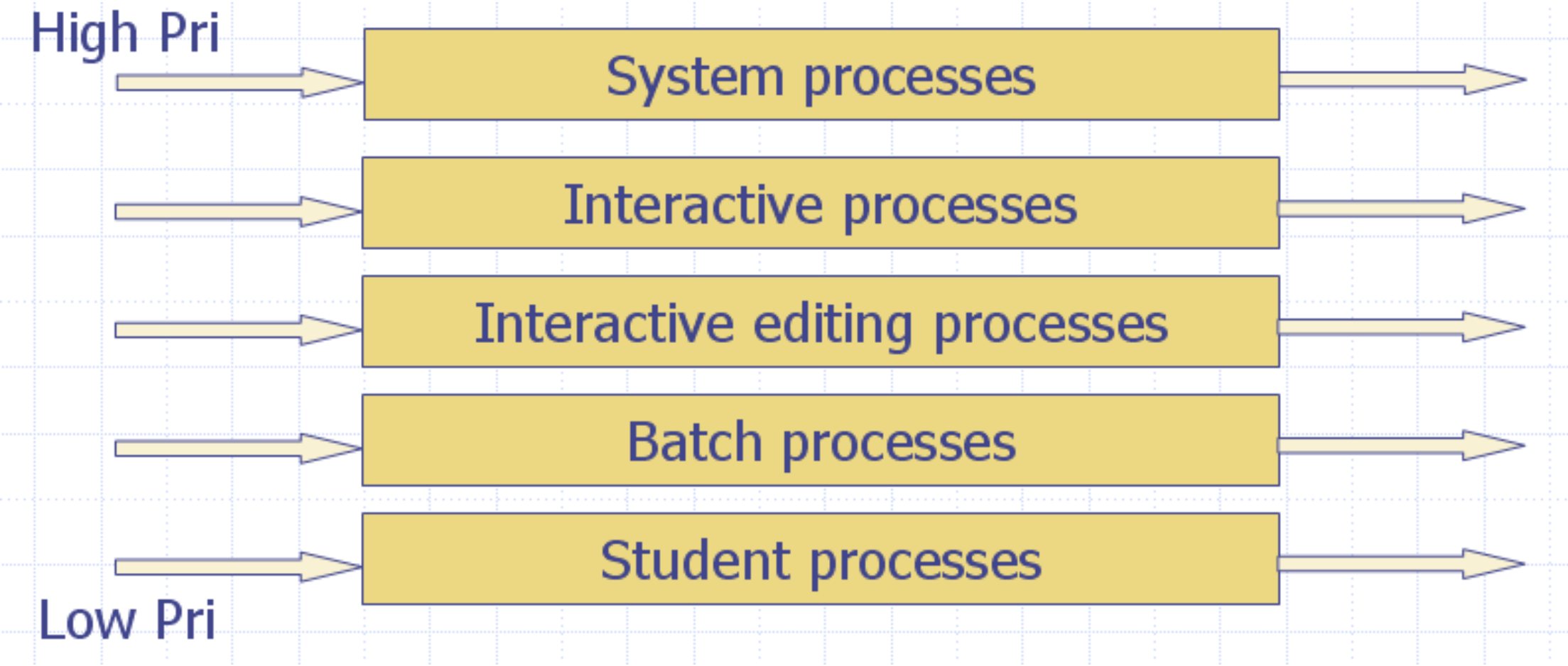
Organizacija vežbi

- **Termini održavanja**
  - **Utorak 8:15, Sala 308**
  - **Sreda 10:15, Sala 313**
- **E-mail: zika@etf.rs**

Promena konteksta - Podsetnik

```
void dispatch() {
        // ...
        // save the context of the currently running process
        // ...

        running->state = Ready;
        Scheduler::put(running);

        running = Scheduler::get();
        running->state = Running;

        // ...
        // restore the context of the new running process
        // ...
}
```

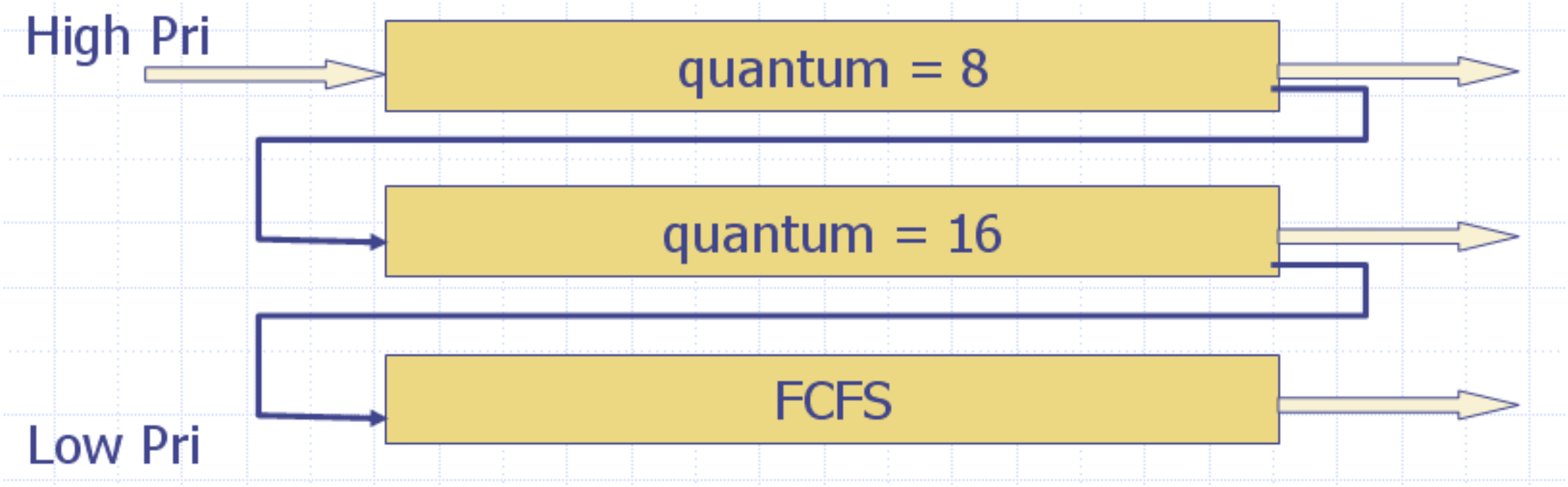
Multilevel Queue Scheduling(MQS)



```
typedef int ThreadType;

class Thread : public Object{
public:     ThreadType type;
// .... };
```

```
class MQScheduler{
public:
        static Queue *system, *interactive,
                *interactiveEditors, *batch, *students;

        static Thread* get() {
                Thread* t = (Thread*) system->get();
                if (t) return t;

                t = (Thread*) interactive->get();
                if (t) return t;

                t = (Thread*) interactiveEditors->get();
                if (t) return t;

                t = (Thread*) batch->get();
                if (t) return t;

                t = (Thread*) students->get();
                return t;
        }

        static void put(Thread* t) { /* ? */ }
};
```

MFQS (*Multilevel Feedback-Queue Scheduling)*

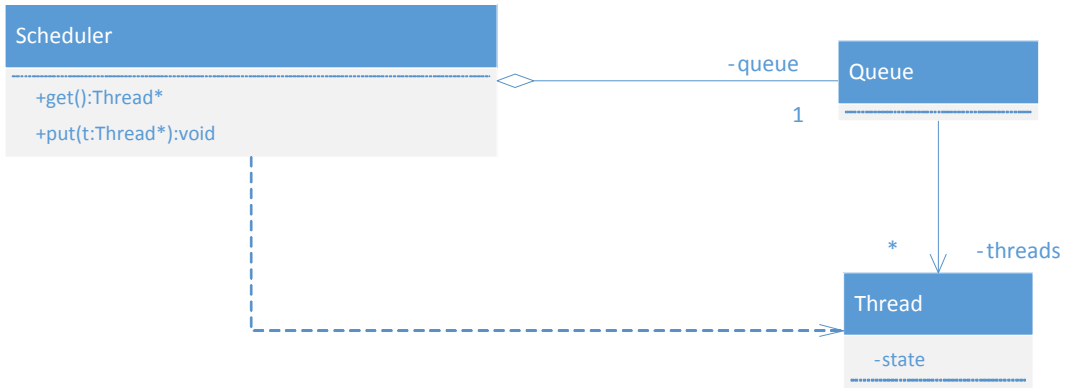


```
class MFQScheduler {
public:
        static Queue *q8, *q16, *fcfs;

        static Thread* get() {
                // …
        }

        static void put(Thread* t) {
                if (t->type == New && t->state == Ready)
                        { t->type = typeQ8; q8->put(t); return; }

                if (t->type == typeQ8 && t->state == Ready)
                        { t->type = typeQ16; q16->put(t); return; }

                if (t->state == Ready)
                        { t->type = typeFCFS; fcfs->put(t); return; }
        }
};
```

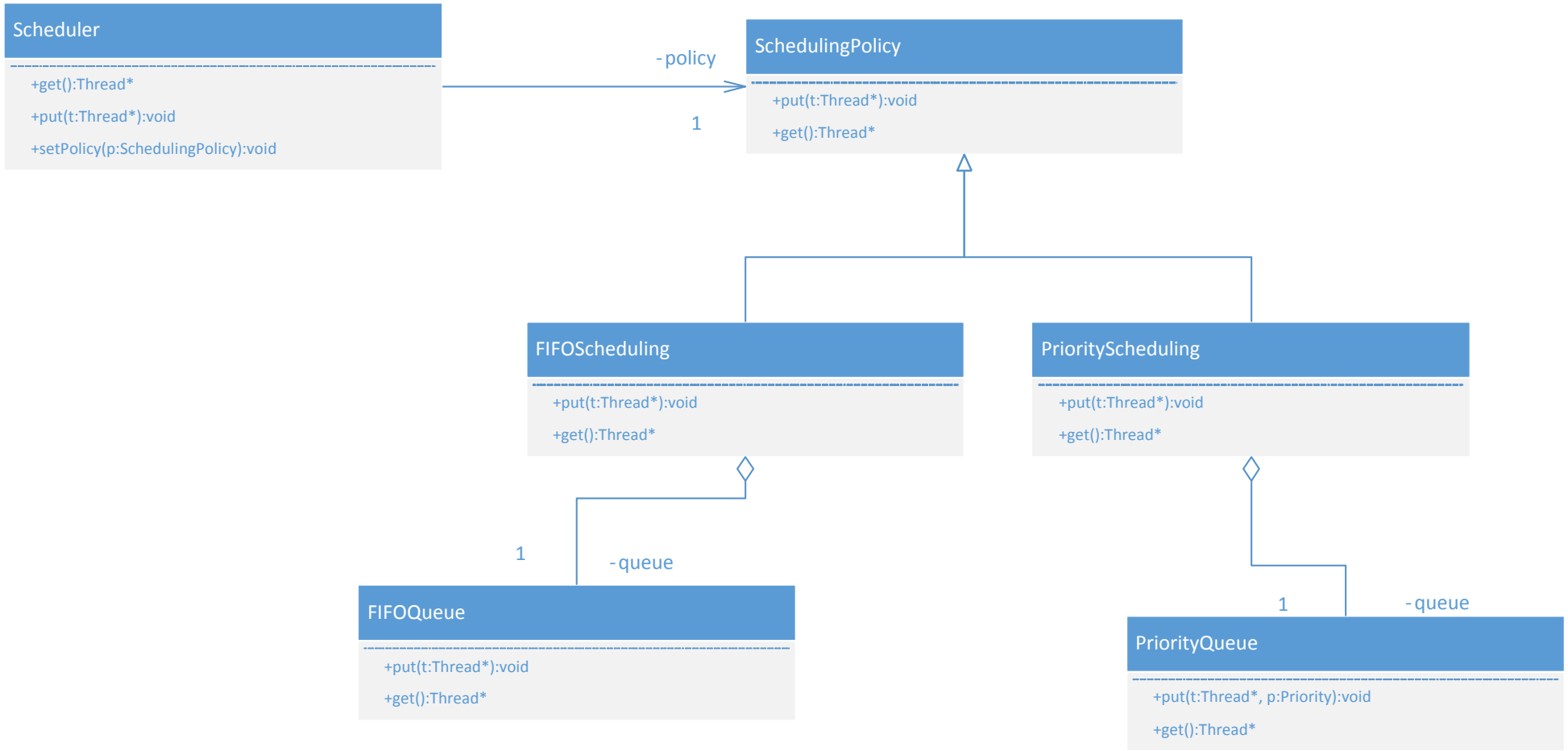
Zadatak

Modelovati i implementirati podsistem za raspoređivanje procesa, koji omogućuje da se algoritam raspoređivanja bira dinamički (u vreme izvršavanja). Predvideti postojanje FIFO algoritma zamene, kao i raspoređivanje po prioritetu.

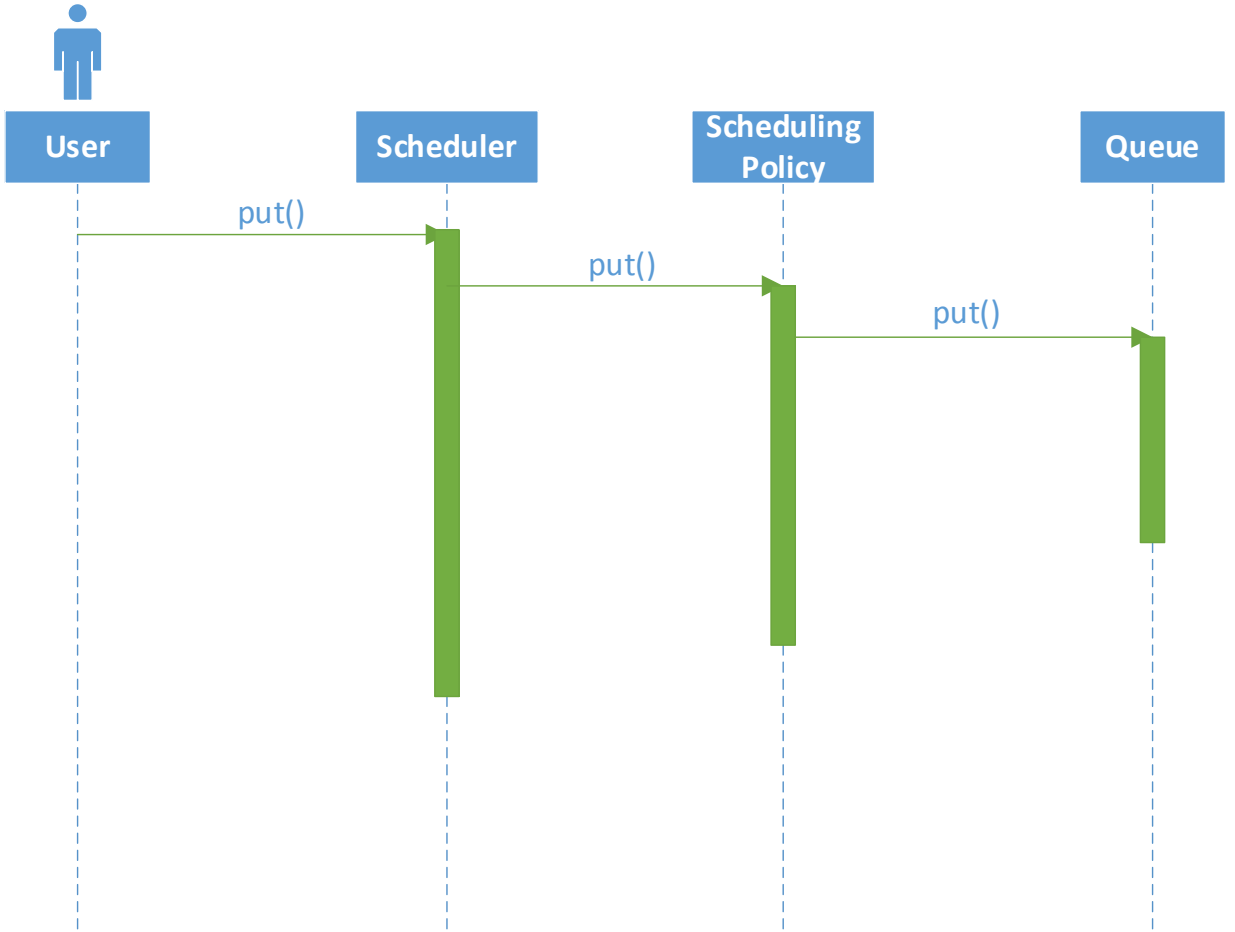
Na visokom nivou



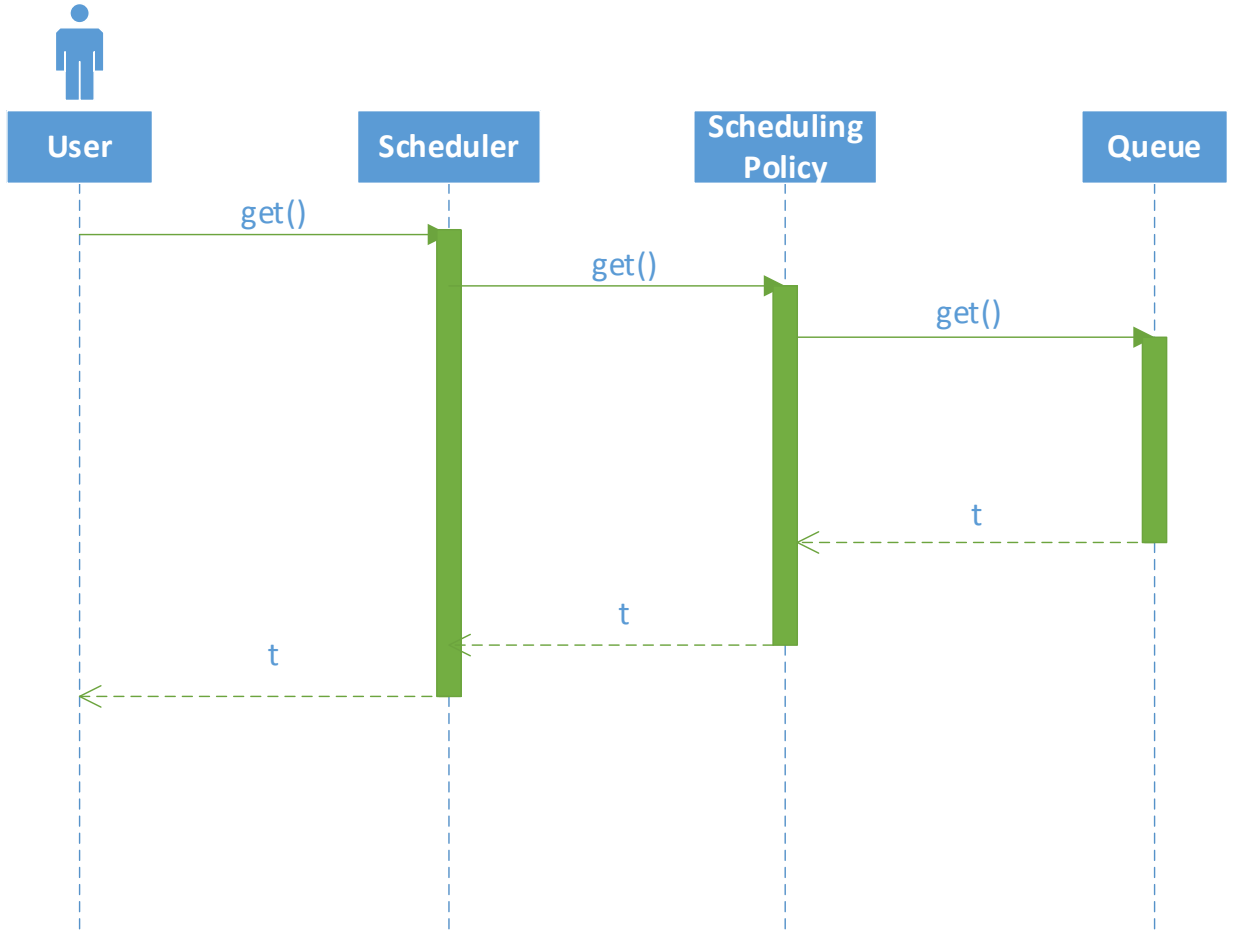
Detaljnije


Scheduler
+get():Thread*
+put(t:Thread*):void
+setPolicy(p:SchedulingPolicy):void
-policy
1
SchedulingPolicy
+put(t:Thread*):void
+get():Thread*
FIFOScheduling
+put(t:Thread*):void
+get():Thread*
PriorityScheduling
+put(t:Thread*):void
+get():Thread*
1
-queue
FIFOQueue
+put(t:Thread*):void
+get():Thread*
1
-queue
PriorityQueue
+put(t:Thread*, p:Priority):void
+get():Thread*

Dijagram sekvence za metodu put

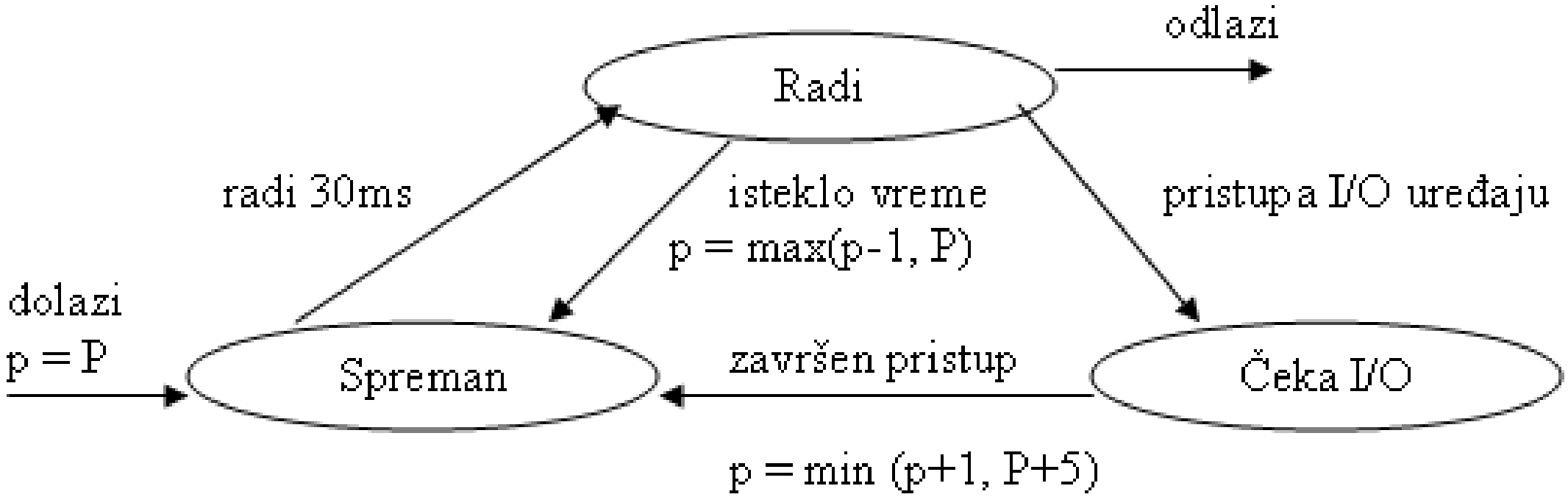


Dijagram sekvence za metodu get



Zadatak

Implementirati algoritam za raspoređivanje procesa prema sledećem dijagramu prelaza:



Veliko P je minimalni prioritet procesa, a malo p je tekući prioritet procesa.

```
class PriorityScheduling{
public:
	static Queue *red;
	static void put(Thread* t) {
		int currP, at = 0;
		if (t->type == New) {
			t->p = t->P;
		}
		else if (t->type == typeCPU){
			t->p = (t->P >= t->p)? t->P : t->p - 1;
		}
		else if (t->type == typeIO) {
			t->p = (t->P+4 < t->p)? t->P + 5 : t->p + 1;
		};
		t->type = typeCPU;
		currP = t->p;
		while (red->getPriority(at) > currP) ++at;
		red->insert(t, at);
	}
	Thread* get() {
		return (red) ? red->remove() : 0;
	}
};
```

Zadatak - Septembar 2006

U nekom operativnom sistemu primenjuje se raspoređivanje procesa sa prioritetima (*Priority Scheduling*), uz starenje (*aging*) radi sprečavanja izgladnjivanja: svaki put kada se neki spreman proces izabere za izvršavanje, ostali u listi spremnih koji još čekaju dobijaju za jedan viši nivo prioriteta. PCB strukture svih procesa su prealocirane i smeštene u statički alociran i dimenzionisan niz `processes`:

```
typedef unsigned short Priority; // Priority: 0 is the lowest
const Priority maxPri = ...; // Max value of priority
const unsigned int maxProc = ...; // Max number of processes

struct PCB {
  int isUsed;  // Is this PCB used for a process (1) or is a free slot (0)?
  int isReady; // Is this process ready?
  Priority curPri; // Current priority
  Priority defPri; // Default priority (set on creation)
  ...
};

PCB processes[maxProc];
```

Realizovati operacije raspoređivača:

```
PCB* getReady (); // Returns the selected process from the „ready list“
void putReady (PCB*); // Puts a process to the „ready list“
```

```
PCB* getReady () {
  Priority mPri = 0;
  PCB* selProc = 0;
  // find the biggest priority
  for (unsigned int i=0; i<maxProc; i++) {
    PCB* p = &processes[i];
    if (!p->isUsed || !p->isReady) continue;
    if (p->curPri>=mPri) {
      mPri = p->curPri;
      selProc = p;
    }
   }
  if (selProc==0) return 0;
  for (unsigned int i=0; i<maxProc; i++) { //aging
    PCB* p = &processes[i];
    if (!p->isUsed || !p->isReady) continue;
    if (p!=selProc && p->curPri<maxPri) p->curPri++;
  }
  return selProc;
}
```

```
void putReady (PCB* p) {
  if (p) {
    p->isReady = 1;
    p->curPri=p->defPri;
  }
}
```

Zadatak - Oktobar 2006

U nekom operativnom sistemu primenjuje se raspoređivanje procesa sa aproksimacijom SJF (*Shortest Job First*) algoritma, uz predviđanje trajanja narednog naleta izvršavanja (*CPU burst*) sa eksponencijalnim usrednjavanjem i $\alpha = ½$. PCB strukture svih procesa su prealocirane i smeštene u statički alociran i dimenzionisan niz `processes`:

```
typedef unsigned int Time; // Execution time in clock ticks

struct PCB {
  int isUsed;  // Is this PCB used for a process (1) or is a
free slot (0)?
  int isReady; // Is this process ready?
  Time prediction; // Prediction of CPU burst duration
  Time lastCPUBurst; // Last actual CPU burst duration
  ...
};

PCB processes[maxProc];
```

Realizovati operacije raspoređivača:

```
PCB* getReady (); // Returns the selected process from the „ready list“
void putReady (PCB*); // Puts a process to the „ready list“
```

Pretpostaviti da je pre poziva funkcije `putReady()` u član `lastCPUBurst` strukture `PCB` datog procesa (koji je upravo izgubio procesor) već upisana vrednost trajanja upravo završenog naleta izvršavanja, kao i da je u članu `prediction` vrednost procene tog naleta; u ovoj funkciji član `prediction` treba ažurirati procenom trajanja narednog naleta izvršavanja.

```
PCB* getReady () {
  Time minTime = ~0;
  PCB* selProc = 0;
  for (unsigned int i=0; i<maxProc; i++) {
    PCB* p = &processes[i];
    if (!p->isUsed || !p->isReady) continue;
    if (p->prediction<=minTime) {
      minTime = p->prediction;
      selProc = p;
    }
  }
  return selProc;
}

void putReady (PCB* p) {
  if (p) {
    p->isReady = 1;
    p->prediction=(p->prediction+p->lastCPUBurst)/2;
  }
}
```

Zadatak - Jun 2010

U nekom sistemu primenjuje se SJF (*shortest job first*) raspoređivanje procesa, uz približnu procenu trajanja narednog naleta izvršavanja eksponencijalnim usrednjavanjem. Procenjeno trajanje naleta izvršavanja je uvek ceo broj u opsegu 0..*T*-1. Razmatraju se tri različite implementacije skupa spremnih procesa:

A) Dvostruko ulančana, neuređena lista.

B) Dvostruko ulančana lista uređena po rastućem redosledu procenjenog trajanja narednog naleta izvršavanja.

C) Niz od *T* ulaza, u svakom ulazu broj *t* je dvostruko ulančana lista (preciznije, glava i rep liste) spremnih procesa sa procenjenim narednim naletom izvršavanja jednakim *t*.

Za svaku od ovih realizacija u donjoj tabeli dati kompleksnost odgovarajuće operacije u funkciji broja spremnih procesa *n*.

Operacija	A	B	C
Dodavanje novog spremnog procesa u skup spremnih			
Uzimanje procesa za izvršavanje (sa njegovim izbacivanjem iz skupa spremnih)			
Dinamička promena procene trajanja narednog naleta izvršavanja procesa koji je u skupu spremnih			

Zadatak - Januar 2006

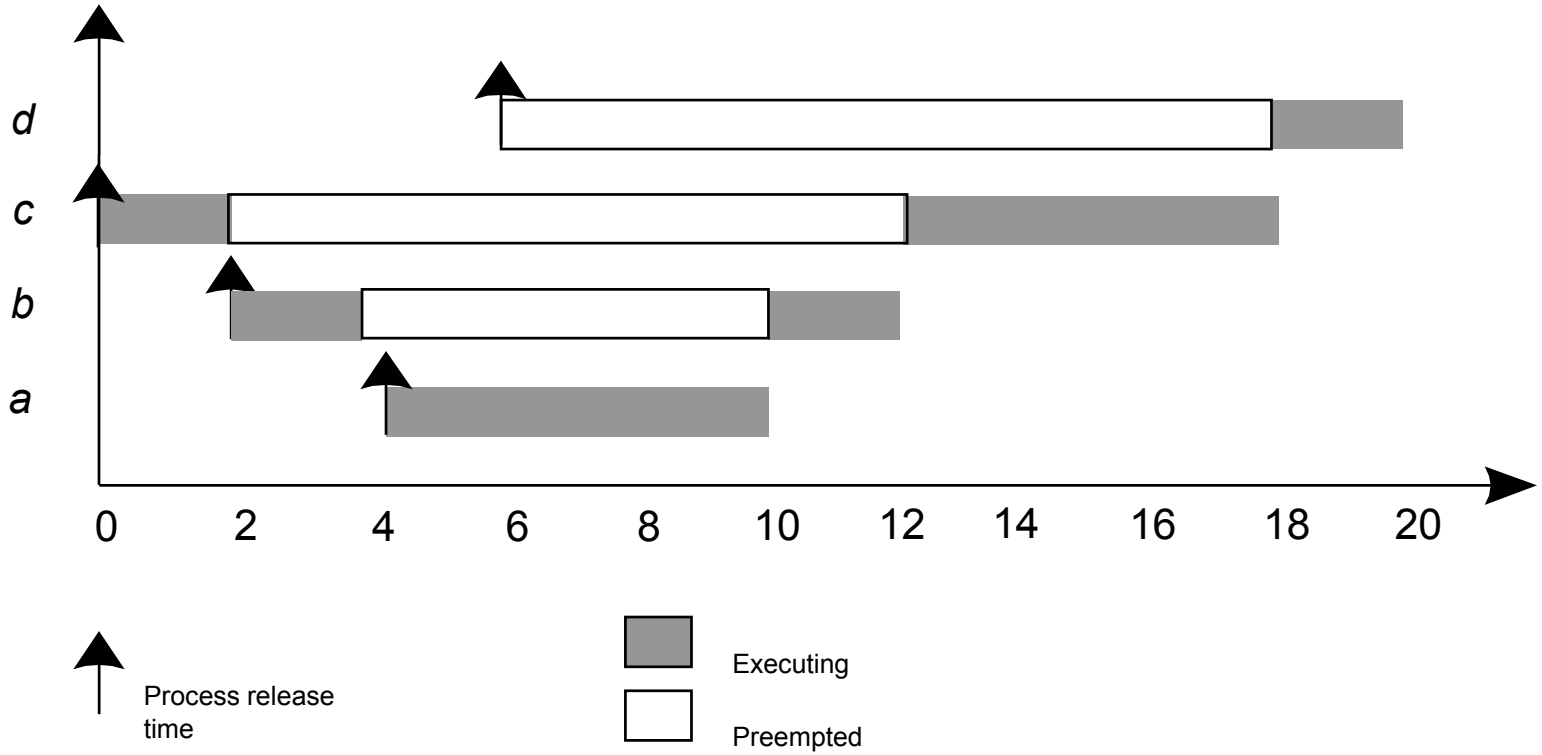
U sistemu se aktiviraju (kreiraju ili postaju spremni) procesi sledećih karakteristika (niži broj označava viši prioritet):

Proces	Prioritet	Trenutak aktivacije	Dužina izvršavanja
A	0	4	6
B	1	2	4
C	2	0	8
D	3	6	2

U tabelu upisati u kom trenutku dati proces počinje svoje (prvo) izvršavanje i u kom trenutku se završava, kao i vreme odziva procesa i ukupno srednje vreme odziva procesa, ako je algoritam raspoređivanja:

Proces	Prioritet	Trenutak aktivacije	Dužina izvršavanja
A	0	4	6
B	1	2	4
C	2	0	8
D	3	6	2

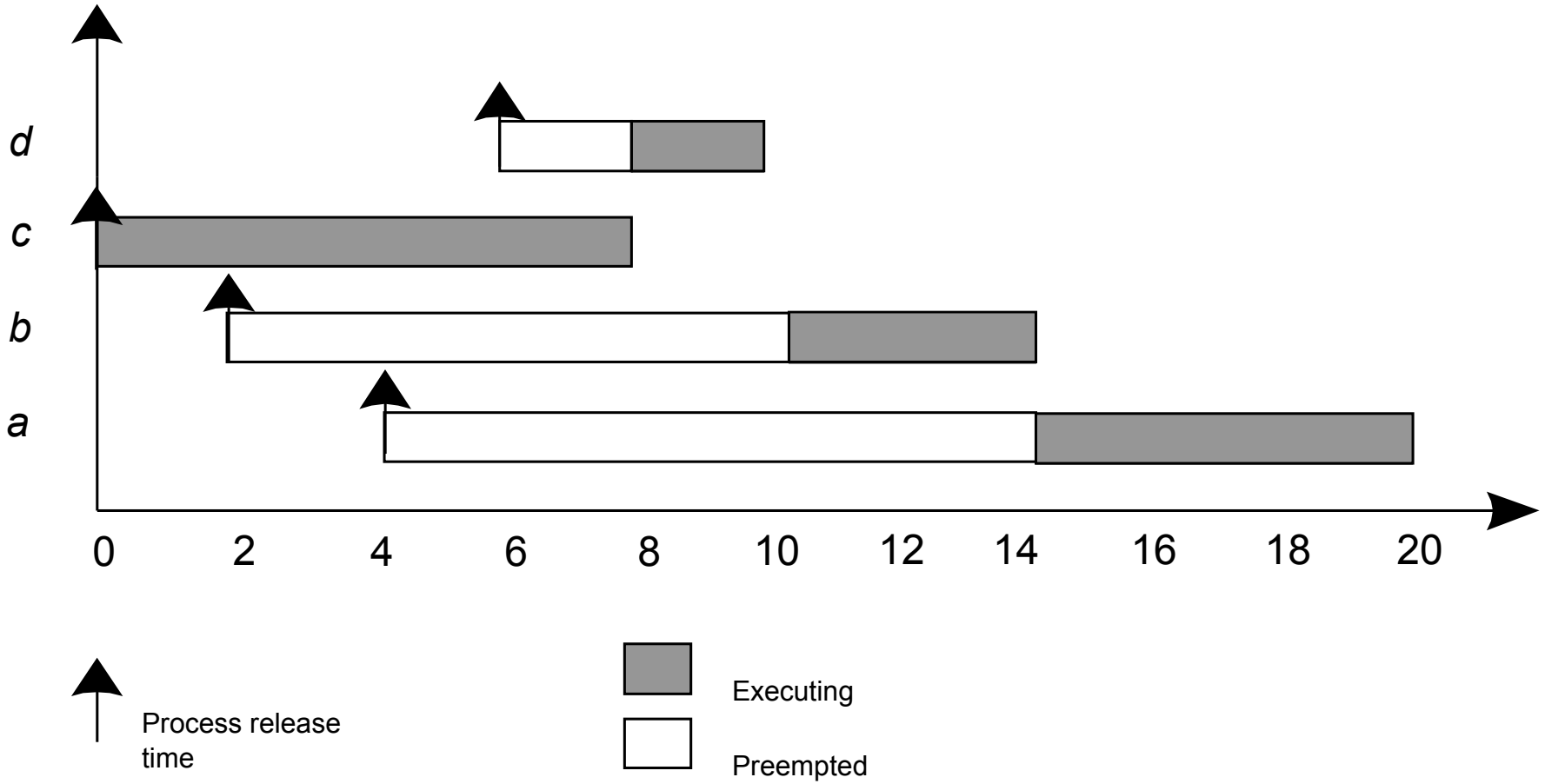
a)(5) po prioritetu sa preuzimanjem (*Preemptive Priority Scheduling*)



Proces	Trenutak prvog izvršavanja	Trenutak završetka	Vreme odziva
A	4	10	6
B	2	12	10
C	0	18	18
D	18	20	14
Srednje vreme odziva:			12

Proces	Prioritet	Trenutak aktivacije	Dužina izvršavanja
A	0	4	6
B	1	2	4
C	2	0	8
D	3	6	2

b)(5) najkraći-posao-prvi bez preuzimanja (*Nonpreemptive SJF*).



Proces	Trenutak prvog izvršavanja	Trenutak završetka	Vreme odziva
A	14	20	16
B	10	14	12
C	0	8	8
D	8	10	4
Srednje vreme odziva:			10

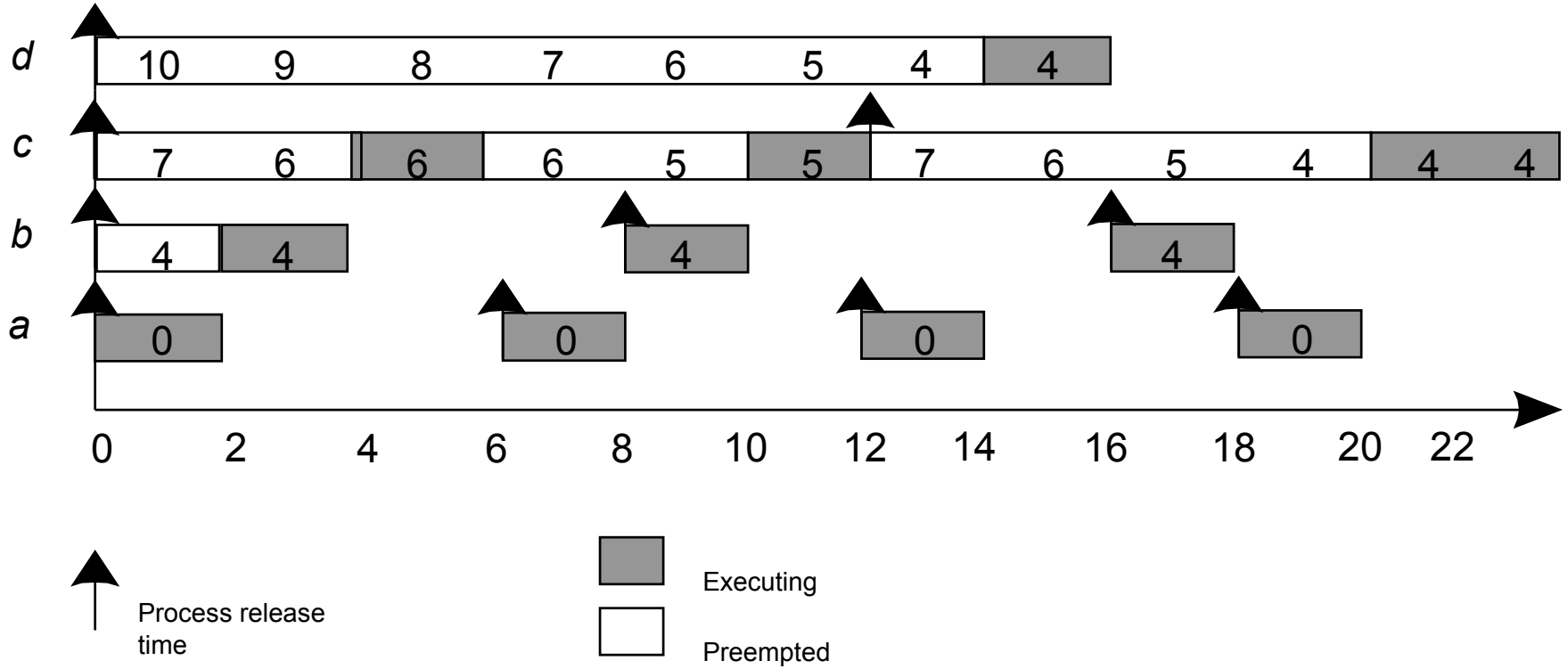
Zadatak - Januar 2009

U nekom sistemu koristi se raspoređivanje procesa po prioritetu (*priority scheduling*), pri čemu se izgladnjivanje sprečava tehnikom starenja (*aging*) na sledeći način. Jezgro periodično, svake druge jedinice vremena, pokreće postupak u kome se svim procesima koji se trenutno nalaze u redu spremnih povećava tekući prioritet za jedan. Na taj način oni procesi koji duže čekaju u redu spremnih vremenom dobijaju sve viši prioritet. Ako neki proces u istom tom trenutku iz reda spremnih treba da dobije procesor, ne menja mu se tekući prioritet jer ne ostaje u redu spremnih; slično, tekući prioritet se ne menja ni procesu koji gubi procesor i dolazi u red spremnih. U ovom sistemu kreirani su sledeći periodični procesi sa datim inicijalnim vrednostima prioriteta prilikom svake periodične aktivacije (niža vrednost označava viši prioritet):

Posmatra se izvršavanje ovih periodičnih procesa od trenutka 0, kada su svi istovremeno aktivirani, do trenutka 24, uz raspoređivanje po prioritetima sa preuzimanjem (*preemptive priority-based scheduling*).

Napisati redosled izvršavanja delova procesa i dužina tih izvršavanja (npr. „A1, B2, C3, D1, C3, ...“), u toku vremenskog intervala [0, 24]. Zanemaruje se vreme promene konteksta i ostalo režijsko vreme.

Proces	Prioritet	Perioda	Vreme izvršavanja u svakoj periodi
A	0	6	2
B	4	8	2
C	7	12	4
D	10	24	2



Odgovor: A2, B2, C2, A2, B2, C2, A2, D2, B2, A2, C4

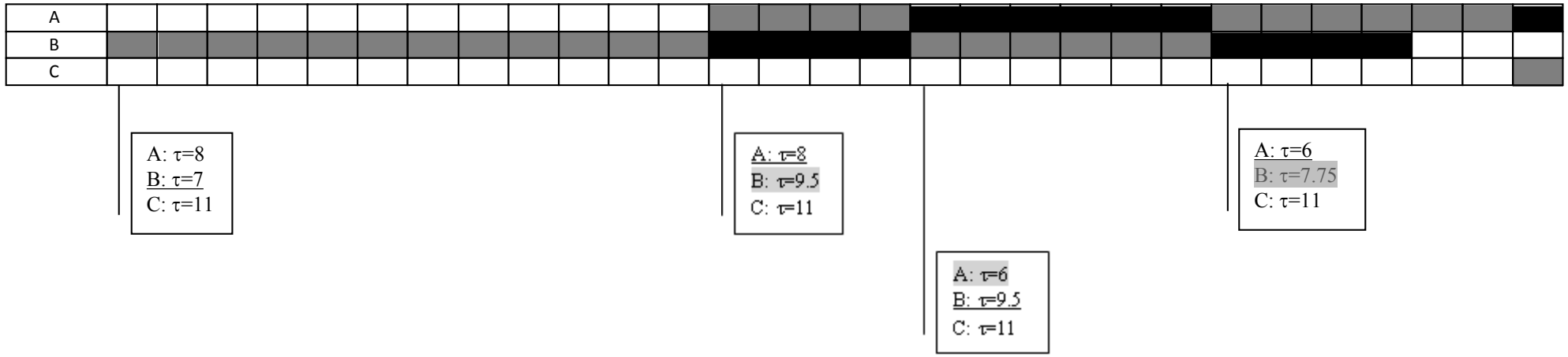
Zadatak – Jun 2006

U nekom operativnom sistemu bez preuzimanja (*non-preemptive*) primenjuje se aproksimacija SJF (*Shortest Job First*) algoritma raspoređivanja procesa, pri čemu se koristi predviđanje sa eksponencijalnim usrednjavanjem sa $\alpha=1/2$. U nekom trenutku u sistemu se nalaze tri spremna procesa A, B i C, čije su vremenske karakteristike date u donjoj tabeli:

Tekuće vrednosti parametara za predviđanje date su u levom delu tabele. Vremena narednih naleta izvršavanja (CPU *burst*) i trajanja I/O operacija (I/O *burst*) data su redom u desnom delu tabele. Na primer, proces A će imati prvi naredni nalet izvršavanja u trajanju od 4 jedinice vremena, pa onda čekati na I/O operaciju koja traje 6 jedinica vremena, pa onda opet imati nalet izvršavanja u trajanju od 6 jedinica vremena, i na kraju čekati na I/O operaciju koja traje 4 jedinice vremena.

Navesti redosled kojim će proces izvršavati CPU nalete ovih procesa. U odgovoru navesti samo redosled kojim će procesor izvršavati CPU nalete procesa, npr. A, B, C, A, B, C, a ispod dati postupak dolaska do odgovora.

	Parametri predviđanja izvršavanja		Naredna vremena naleta izvršavanja (CPU *burst*) i I/O operacija (I/O *burst*)			
Proces	τ_n	CPU *burst* t_n	CPU *burst* (t_{n+1})	I/O *burst*	CPU *burst* (t_{n+2})	I/O *burst*
A	8	8	4	6	6	4
B	2	12	12	4	6	4
C	12	10	8	4	12	4



Legenda:
U okvirima:
Podvučeno: proces koji je izabran za izvršavanje
Sivo: blokiran proces
U tabeli:
Sivo polje: proces se izvršava
Crno polje: proces blokiran (čeka I/O operaciju)

Zadatak

U nekom sistemu koristi se *Multilevel Feedback-Queue Scheduling* (MFQS), uz dodatni mehanizam koji ga približava ponašanju SJF algoritma. Svakom procesu pridružena je procena trajanja narednog naleta izvršavanja τ koja je ceo broj.

- Postoje tri reda spremnih procesa: HP (*High Priority*), MP (*Medium Priority*) i LP (*Low Priority*).
- Globalni algoritam raspoređivanja je po prioritetu, s tim da HP ima najviši, a LP najniži prioritet.
- Raspoređivanje u svim redovima je je *Round-Robin* (RR), samo sa različitim vremenskim kvantumom koji se dodeljuje procesima.
- Proces se smešta u red prema proceni τ: ako je $\tau \leq 5$, smešta se u HP, ako je $5 < \tau \leq 10$, smešta se u MP, inače se smešta u LP.

Klasa `Scheduler`, čiji je interfejs dat dole, implementira opisani raspoređivač spremnih procesa. Implementirati ovu klasu tako da i operacija dodavanja novog spremnog procesa `put()` i operacija uzimanja spremnog procesa koji je na redu za izvršavanje `get()` budu ograničene po vremenu izvršavanja vremenom koje ne zavisi od broja spremnih procesa (kompleksnost $O(1)$). U slučaju da nema drugih spremnih procesa, treba vratiti proces na koga ukazuje `idle` (to je uvek spreman proces najnižeg prioriteta). U strukturi `PCB` polje `tau` tipa `int` predstavlja procenu τ, a polje `next` pokazivač tipa `PCB*` koji služi za ulančavanje struktura PCB u jednostruke liste. Operacija get() treba da postavi polje timeslice u PCB odabranog procesa na vrednost konstante koja odgovara vremenskom kvantumu za red iz koga je proces izvađen. Vrednost tau procesa koji se smešta pomoću put() je već izračunata spolja pre poziva te operacije.

```
const int timesliceHP = ..., timesliceMP = ..., timesliceLP = ...;
extern PCB * const idle;

class Scheduler {
public:
  Scheduler ();
  PCB* get ();
  void put (PCB*);
};
```

```
const int timesliceHP = ..., timesliceMP = ..., timesliceLP = ...;
extern PCB * const idle;

class Scheduler {
public:
  Scheduler ();
  PCB* get ();
  void put (PCB*);
private:
  PCB* head[3];  // 0-HP, 1-MP, 2-LP
  PCB* tail[3];
  static const int timeslice[3];
};

const int Scheduler::timeslice[] = {timesliceHP,timesliceMP,timesliceLP};

Scheduler::Scheduler () {
  for (int i=0; i<2; i++)
    head[i]=tail[i]=0;
}

PCB* Scheduler::get () {
  for (int i=0; i<2; i++)
    if (head[i]) {
      PCB* ret = head[i];
      head[i] = head[i]->next;
      if (head[i]==0) tail[i]=0;
      ret->next = 0;
      ret->timeslice=timeslice[i];
      return ret;
    }
  return idle;
}
```

```
void Scheduler::put (PCB* pcb) {
  if (pcb==0) return; // Exception!
  int i=0;
  if (pcb->tau<=5) i=0;
  else
  if (pcb->tau<=10) i=1;
  else i=2;
  pcb->next = 0;
  if (tail[i]==0)
    tail[i] = head[i] = pcb;
  else
    tail[i] = tail[i]->next = pcb;
}
```

Zadatak

U nekom sistemu koristi se *Multilevel Feedback-Queue Scheduling* (MFQS) na sledeći način:

- Postoje tri reda spremnih procesa: HP (*High Priority*), MP (*Medium Priority*) i LP (*Low Priority*).
- Globalni algoritam raspoređivanja je po prioritetu, s tim da HP ima najviši, a LP najniži prioritet.
- Raspoređivanje u svim redovima je *Round-Robin* (RR), samo sa različitim vremenskim kvantumom koji se dodeljuje procesima.
- Procesima koji se uzimaju iz HP dodeljuje se vremenski kvantum 4, onima koji se uzimaju iz MP vremenski kvantum 8, a onima iz LP kvantum 16.
- Proces koji je tek postao spreman (bio je blokiran ili je tek kreiran) smešta se u red prema proceni dužine svog sledećeg naleta izvršavanja (*CPU burst*) na sledeći način: ako je procena $\tau \leq 4$, stavlja se u red HP; ako je procena $4 < \tau \leq 8$, stavlja se u red MP; inače se stavlja u red LP.
- Procena dužine sledećeg naleta izvršavanja vrši se eksponencijalnim usrednjavanjem sa koeficijentom $\alpha = 1/2$ i pamti se kao ceo broj (vrši se odsecanje pri izračunavanju).
- Proces kome je istekao vremenski kvantum smešta se u MP ako je prethodno bio uzet iz HP, odnosno u LP ako je prethodno bio uzet iz MP ili LP.

Posmatra se jedan proces koji ima sledeće nalete izvršavanja (označeni sa C i dužinom trajanja naleta) i ulazno/izlazne operacije (označene sa I/O), i sa početnom procenom $\tau = 3$:

C13, I/O, C2, I/O, C9, I/O, C12, I/O, C3

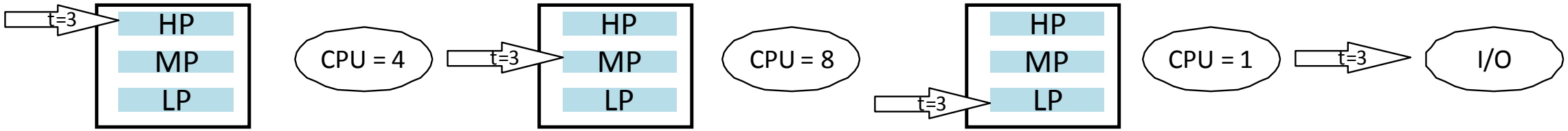
a) Dati oznake redova spremnih procesa (HP, MP, LP) u koje je ovaj proces redom stavljan, i to za svako stavljanje procesa u neki od redova spremnih (odgovor dati u obliku npr. HP, MP, LP, LP, LP, ...)

Odgovor:_____

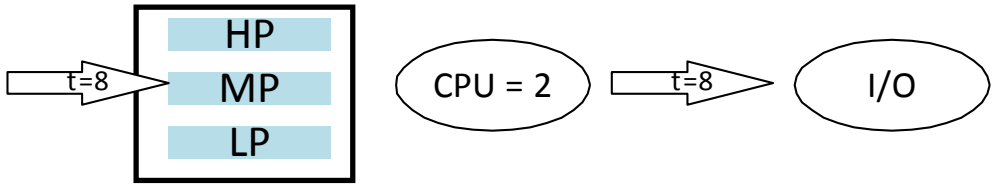
b) Koja je vrednost procene dužine sledećeg naleta izvršavanja τ nakon ove sekvence?

Odgovor: τ = _____

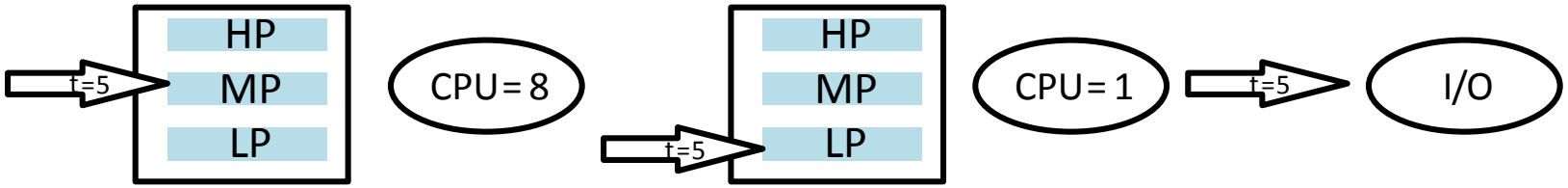
C13, I/O, C2, I/O, C9, I/O, C12, I/O, C3



t=(3+13)/2=8

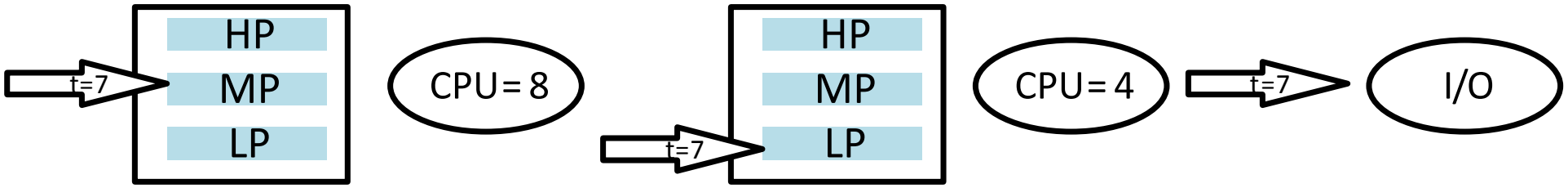


t=(8+2)/2=5

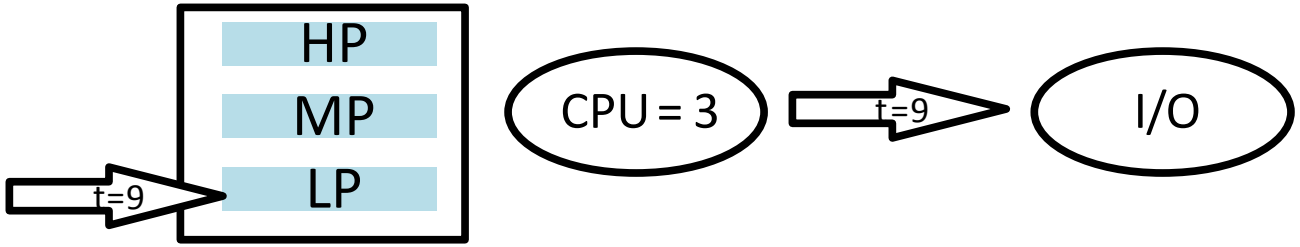


C13, I/O, C2, I/O, C9, I/O, C12, I/O, C3

t=(5+9)/2=7



t=(7+12)/2=9



t=(9+3)/2=6

Zadatak – Januar 2017

U nekom sistemu klasa `Scheduler`, čija je delimična definicija data dole, realizuje raspoređivač spremnih procesa za simetrični multiprocesorski sistem sa tzv. *afinititetom* (engl. *affinity*) procesa: teži se da dati proces dobije isti procesor na kom se prethodno već izvršavao, kako bi se smanjilo kašnjenje zbog promašaja u procesorskom kešu koje bi postojalo kada bi se on izvršavao na drugom procesoru. Osim toga, sistem dodeljuje dati procesor onom procesu dodeljenom tom procesoru, koji je od početka svog tekućeg naleta izvršavanja (tj. od kako je došao u red spremnih npr. iz stanja suspenzije) imao najkraće ukupno vreme izvršavanja na procesoru tokom tekućeg naleta izvršavanja (engl. *CPU burst*).

- Postoji *P* simetričnih (jednakih i ravnopravnih) procesora, pri čemu je *P* konfiguraciona konstanta.
- Za svaki od *P* procesora postoji poseban red spremnih procesa, uređenih po ukupnom vremenu izvršavanja procesa na procesoru u tekućem naletu izvršavanja; ovo vreme čuva se u polju `execTime` strukture PCB svakog procesa.
- Kada stavlja dati proces u red spremnih operacijom `Scheduler::put()`, u drugom argumentu `elapsed` ove operacije, sistem dostavlja vreme koje se proces izvršavao na procesoru pre nego što ga je izgubio; ukoliko je proces izgubio procesor zbog isteka vremenskog kvanta ili nekog prekida, ova vrednost jednaka je stvarnom vremenu koje je proteklo od kada je proces dobio procesor; ukoliko pak proces dolazi iz stanja suspenzije ili je tek aktiviran, ova vrednost je 0.
- Ako proces dolazi u red spremnih jer je izgubio procesor (`elapsed!=0`), stavlja se u red spremnih istog procesora na kom se već izvršavao. Taj procesor zapisan je u polju `affinity` strukture PCB.

- Ako proces dolazi u red spremnih iz stanja suspenzije ili je tek aktiviran (`elapsed==0`), stavlja se u red spremnih onog procesora koji ima najmanje spremnih procesa u svom redu, radi raspodele opterećenja procesora (engl. *load balancing*).
- Ukoliko je red spremnih procesa prazan, operacija `Scheduler::get()` treba da vrati 0.
- U strukturi PCB postoji polje `next` kao pokazivač tipa `PCB*` koji služi za ulančavanje struktura PCB u jednostruke liste.

Realizovati u potpunosti klasu Scheduler.

```
class Scheduler {
public:
 Scheduler ();
 PCB* get (int proc); // Get the process for the given processor
 void put (PCB*, unsigned long elapsed);
private:
 static const int P; // Number of processors
 …
};
```

```
class Scheduler {
public:
 Scheduler ();
 PCB* get (int proc);
 void put (PCB*, unsigned long elapsed);
private:
 static const int P;
 PCB* head[P]; // Heads of processors' ready lists
 unsigned long size[P]; // Number of processes in each ready list
};
Scheduler::Scheduler () {
 for (int i=0; i<P; i++) head[i]=size[i]=0;
}
```

```
void Scheduler::put (PCB* pcb, unsigned long elapsed) {
 if (pcb==0) return; // Exception!
 int proc = pcb->affinity; // Processor to schedule on
 if (elapsed==0) {
   pcb->execTime = 0;
   unsigned long minSize = size[0];
   proc = 0;
   for (int p=1; p<P; p++)
     if (size[p]<minSize) {
       proc = p;
       minSize = size[p];
     }
   pcb->affinity = proc;
 }
```

```
// Put pcb in proc's queue:
 pcb->execTime += elapsed;
 PCB *prev=0, *cur=head[proc];
 while (cur && cur->execTime<=pcb->execTime) {
   prev = cur; cur = cur->next;
 }
 if (prev==0) {
   pcb->next = head[proc];
   head[proc] = pcb;
 } else {
   prev->next = pcb;
   pcb->next = cur;
 }
 size[proc]++;
}
```

```
PCB* Scheduler::get (int proc) {
 PCB* ret = head[proc];
 if (ret) {
   head[proc] = ret->next;
   ret->next = 0;
   size[proc]--;
 }
 return ret;
}
```