CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Semaphores

Hardware support for synchronization provided by `test_and_set` and `compare_and_swap`, but you will not typically use these directly

Support for locking is provided by mutexes

More powerful synchronization primitive is the **semaphore**

Semaphore is an integer that is accessed only through two *atomic* operations:

- `wait()`
- `signal()`

Confusingly, you will often see `wait` referred to as `P` and `signal` referred to as `V` because the terms were originally introduced in Dutch

`wait` is a "wait and decrement", and `signal` is an increment

```
wait(S) {
    while (S <= 0 )
        ; // busy wait
    S--;
}

signal(S) {
    S++;
}
```

Once again, these operations must be *atomic*

These operations are the *only way* to access the semaphore, whether for writing *or* reading

How is this useful? Several ways

Binary semaphore is always either 1 or 0

This type of semaphore can be used as replacement for mutex

Binary semaphore can be initialized to 1, then processes `wait` on semaphore before entering critical region and `signal` when they leave

Counting semaphore can be used to control access to finite resource (think producer-consumer problem)

Initialize semaphore to number of available instances of resource

Processes must `wait` before taking a resource and `signal` when finished with it

When semaphore is 0, all further requests for resource will block requesting process until another process `signals`

Also useful for synchronization

Assume process `P2` must wait for process `P1` to finish some work before `P2` can start its own work

Initialize semaphore to 0, `P2` `waits` on semaphore, and `P1` `signals` when finished with work

How are semaphores implemented?

Want to avoid busy waiting implied by earlier definition (repeated here)

```
wait(S) {
    while (S <= 0 )
        ; // busy wait
    S--;
}
```

Need to modify `wait` and `signal` operations (though they will do same thing conceptually)

When `wait` is called, if semaphore cannot be decremented further, process blocks and is added to waiting queue for semaphore

Process will not be scheduled again until resource is available

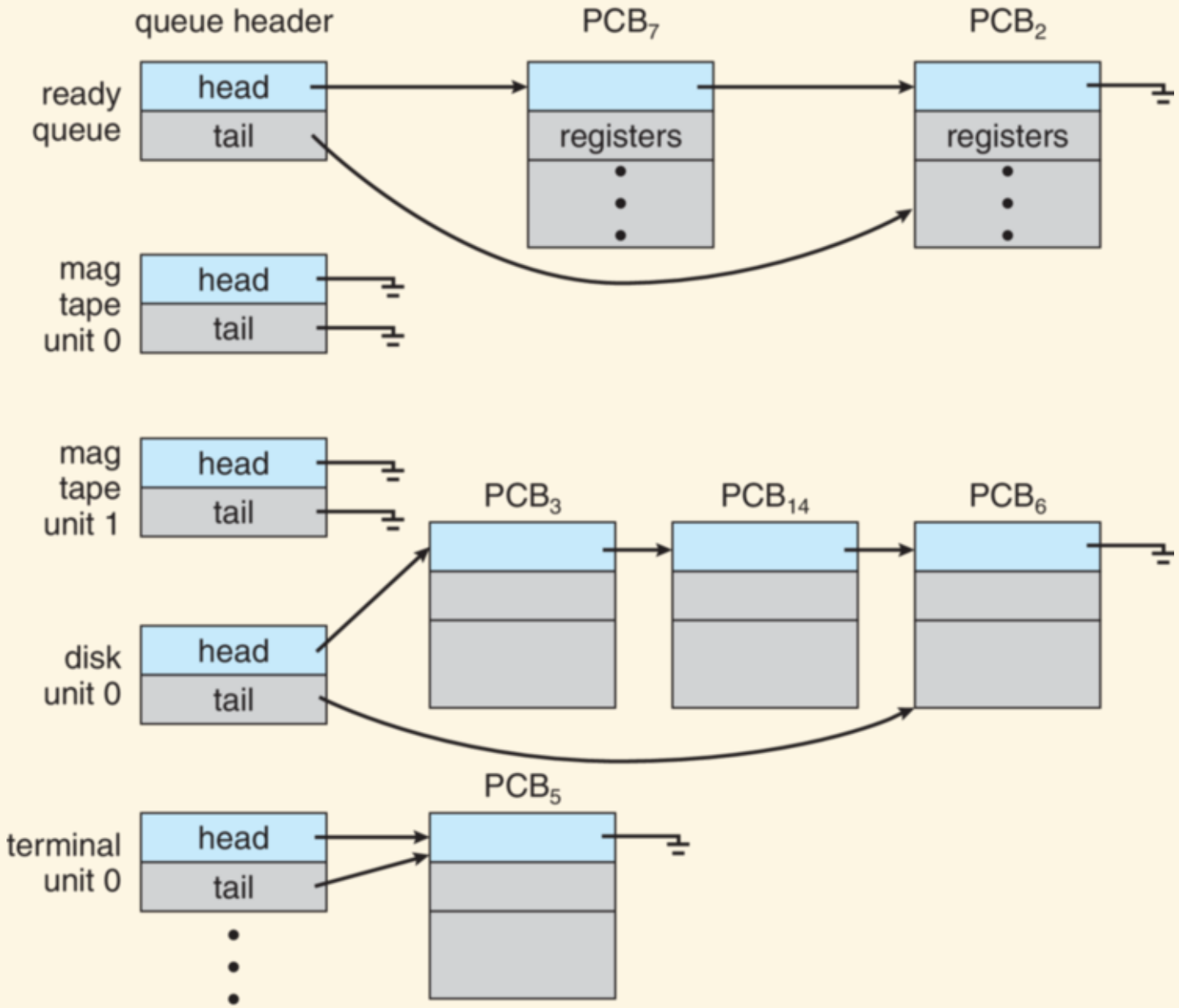
Eliminates busy waiting

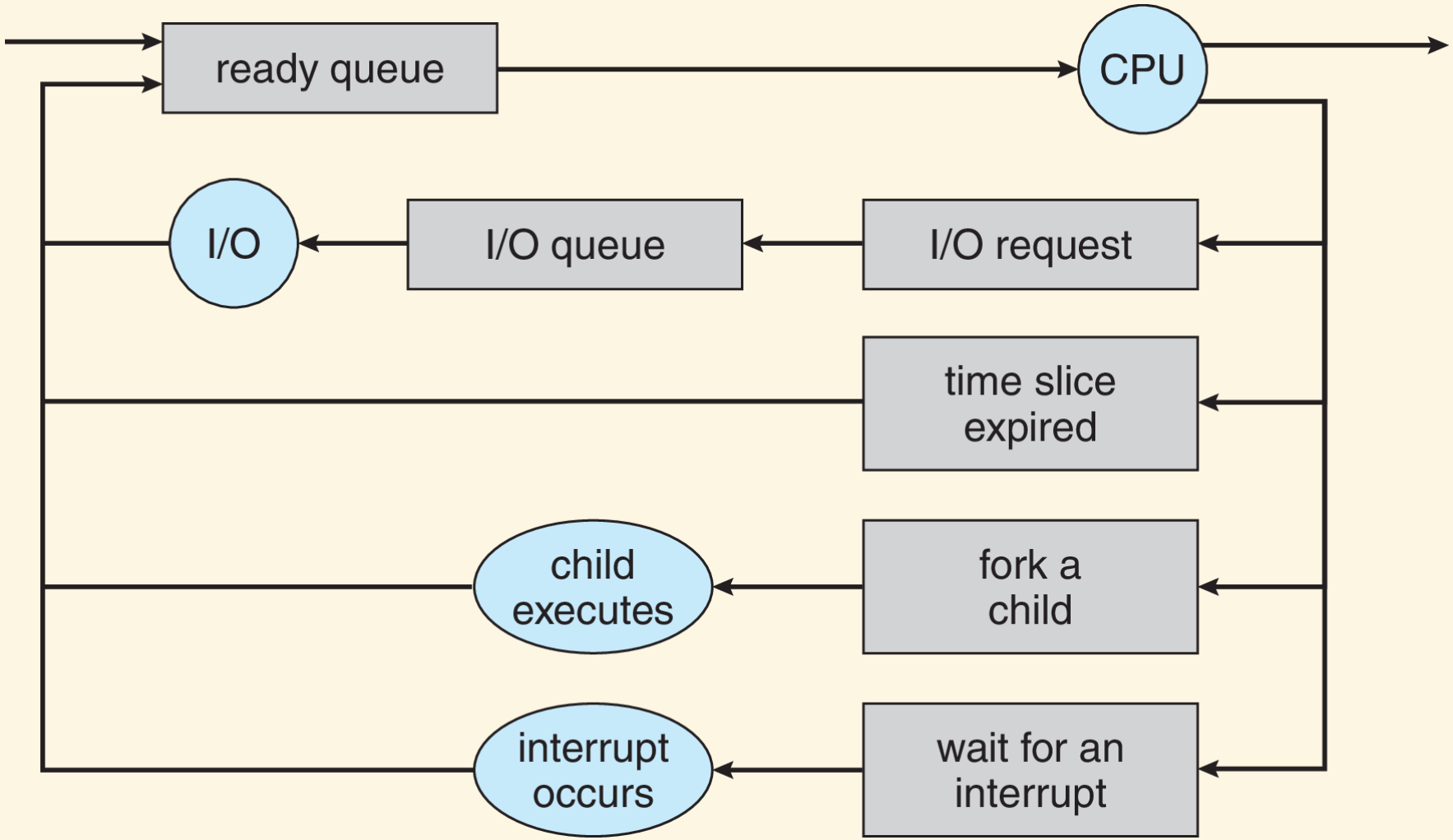
New structure for semaphore

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

New wait operation

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```


queue header
PCB7
PCB2
ready queue
head
tail
registers
registers
mag tape unit 0
head
tail
mag tape unit 1
head
tail
PCB3
PCB14
PCB6
disk unit 0
head
tail
PCB5
terminal unit 0
head
tail


ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

When `signal` is called, one process waiting on semaphore can be woken up (moved to ready state)

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Note that value of semaphore can now be negative. Negative value represents number of waiting processes

`block` and `wakeup` in semaphore definitions are system calls

Give informtaion to CPU scheduler about which processes are ready

`block` and `signal` must be executed *atomically*

No single hardware operation exists to `wait` or `signal`

Implementation of semaphore is itself a critical-section problem

Critical-section problem for implementing semaphores can be solved using lower-level primitives as seen in previous lectures

Typically implemented using busy waiting

We previously said we were trying to avoid busy waiting -- what gives?

Busy waiting is done while waiting for *semaphore code itself*, not for critical section that semaphore is used to protect

Semaphore code

- is relatively short
- will not often be in conflict

```
wait(semaphore *S) {
    // if another process is in same semaphore, busy wait here

    // this is the critical section of semaphore implementation
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }

    // exit critical section
}
```

Old way

```
wait() // busy waiting here while another
       // process completes critical section
/* critical section of program */
signal()
```

New way

```
wait() // Brief busy wait if another process is
       // calling wait() or signal() on this semaphore.
       // Then, sleep until another process completes
       // critical section

/* critical section of program */
signal()
```

Last implementation note -- processes should be removed from list in FIFO (or at least not LIFO) order

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

If processes are removed in LIFO order, process waiting on semaphore can block indefinitely

We call this issue **starvation**

Last thoughts

Important to separate how we *use* semaphores from how they are *implemented*

Remember: a semaphore is conceptually an integer that we access only through `wait` and `signal`

We discussed how semaphores can be used for

- mutual exclusion
- access to limited resources
- synchronization

Semaphores may be implemented with busy waiting or with a waiting queue, but that doesn't affect how we can use them