

Kernel Support for Distributed Systems

Sape J. Mullender

Huygens Systems Research Laboratory
Universiteit Twente
Enschede





Function of an operating system

- define a virtual machine
- realize the virtual machine interface on the physical machine
- multiplex several virtual machines on one physical or virtual machine (IBM VM/370)
- protect virtual machines from interfering with each other

Virtual Machine Interface



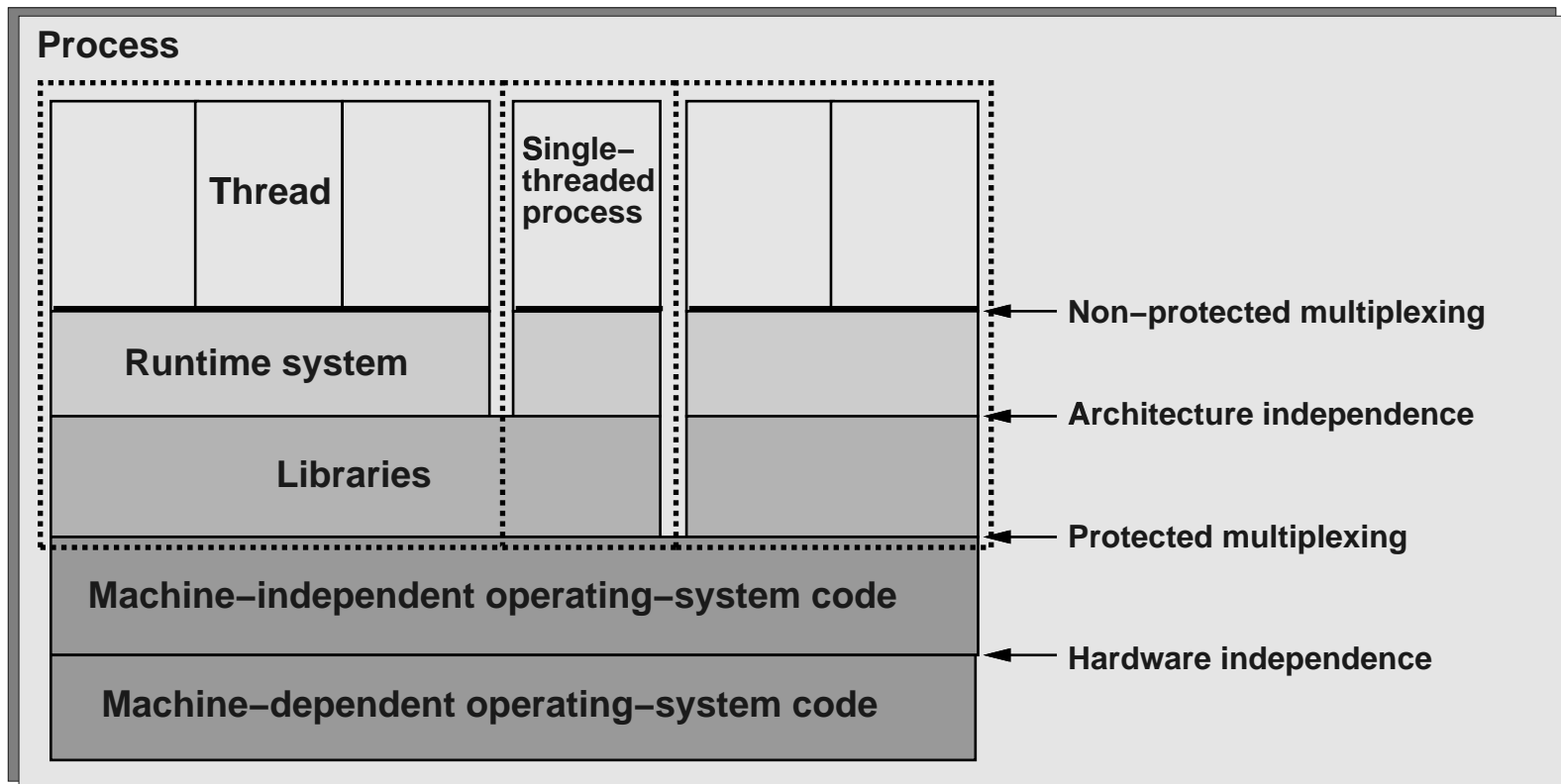
Consists of

- a subset of the physical machine's instructions
- new instructions (system calls)

Layers of Virtual Machines



It is useful to view an operating system as several layers of virtual machines





So how does distributed computing affect an operating system?



The operating system must support applications that can manage *indeterminism*; applications must deal with multiple input streams and it is not known which stream will deliver the next datum.



- Network ─ multiple streams:
 - Expected messages
 - Unsolicited messages
- Timers ─ multiple uses:
 - Protocol related: piggyback, retransmission, latency
 - Application related: calendar, checkpoint/backup, ...
- I/O ─ multiple devices:
 - Keyboard, mouse
 - Storage system
 - cameras, printers, scanners, audio devices, ...

Not just distributed systems



Indeterministic I/O is not confined to distributed applications: Most interactive applications must deal with it too: mouse events, keyboard events and window events (e.g., resize, iconify).

Events and State



Events cause the *state* changes in a (distributed) application. State changes must be synchronized, so an important task of an operating system that supports nondeterministic/concurrent applications is providing synchronization mechanisms.



Various mechanisms:

- *Mutex*

acquire(lock) - release(lock)

- *Semaphore*

P(sem) - V(sem)

- *Monitor*

Implementation



Needs *test-and-set* or a variation of it:

```
Bool TaS(Bit *lock) {
    /* Start atomic operation */
    if (*lock == 0) {
        *lock = 1;
        /* End atomic operation */
        return True;
    }
    /* End atomic operation */
    return False;
}
```



Most difficult on shared-memory multiprocessors.

- Needs a coherent cache (or the ability to turn off caching)
- Needs (one or more) instructions that
 1. lock the memory bus
 2. read a value
 3. write a value
 4. unlock the memory busin such a way that concurrent access from another process is not possible

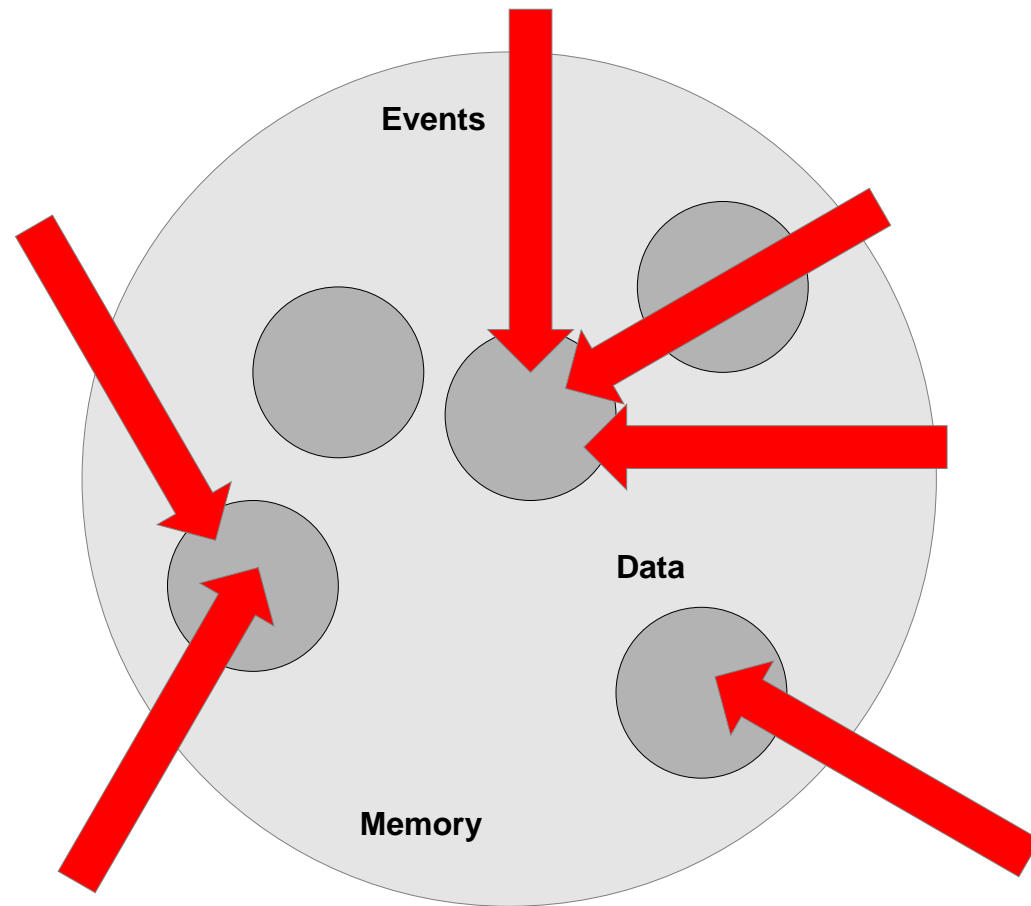
Example



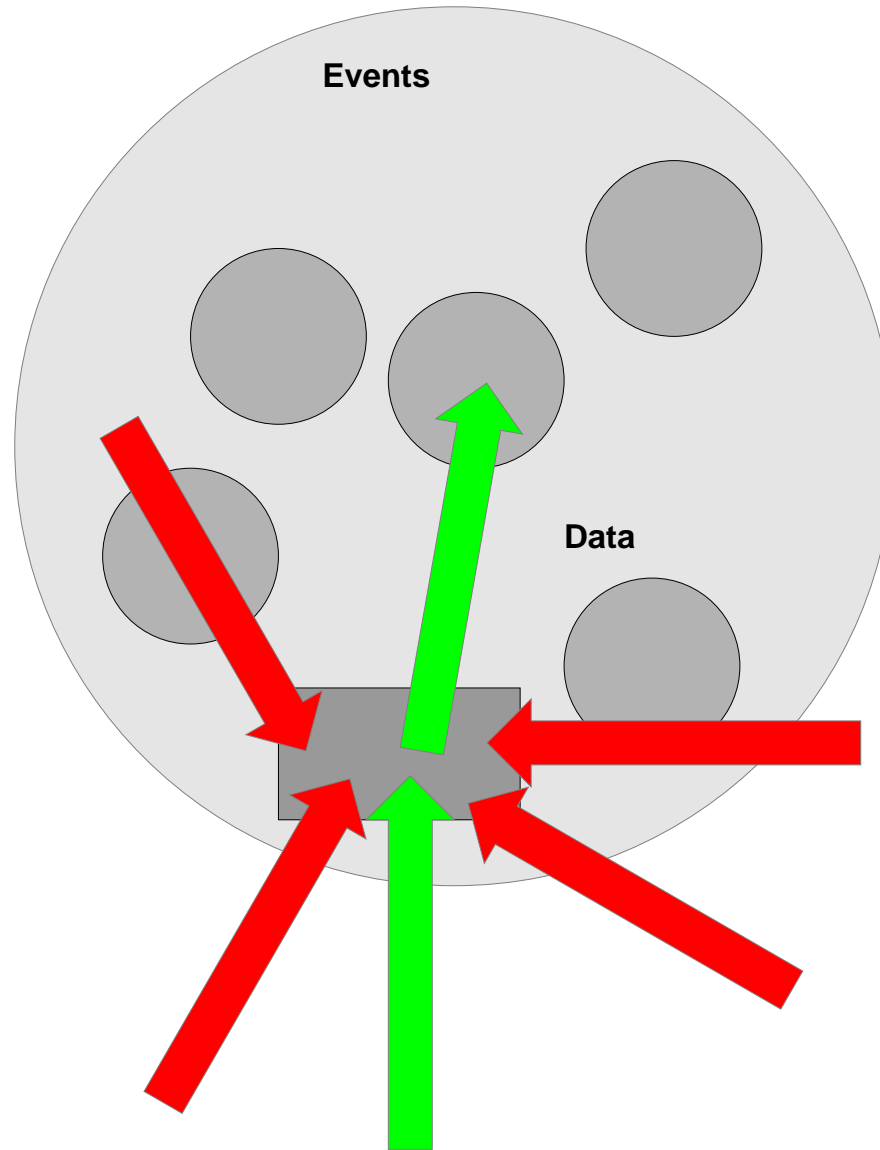
386 family:

```
_tas:
    movl    $0xdeaddead,%eax
    movl    4(%esp),%ecx
    xchgl   %eax,(%ecx) /* atomically exchange values */
    ret     /* 0: success, '0xdeaddead': failure */
```

Structuring Concurrent Applications



Event Loop (Single Monitor)



Event Loop



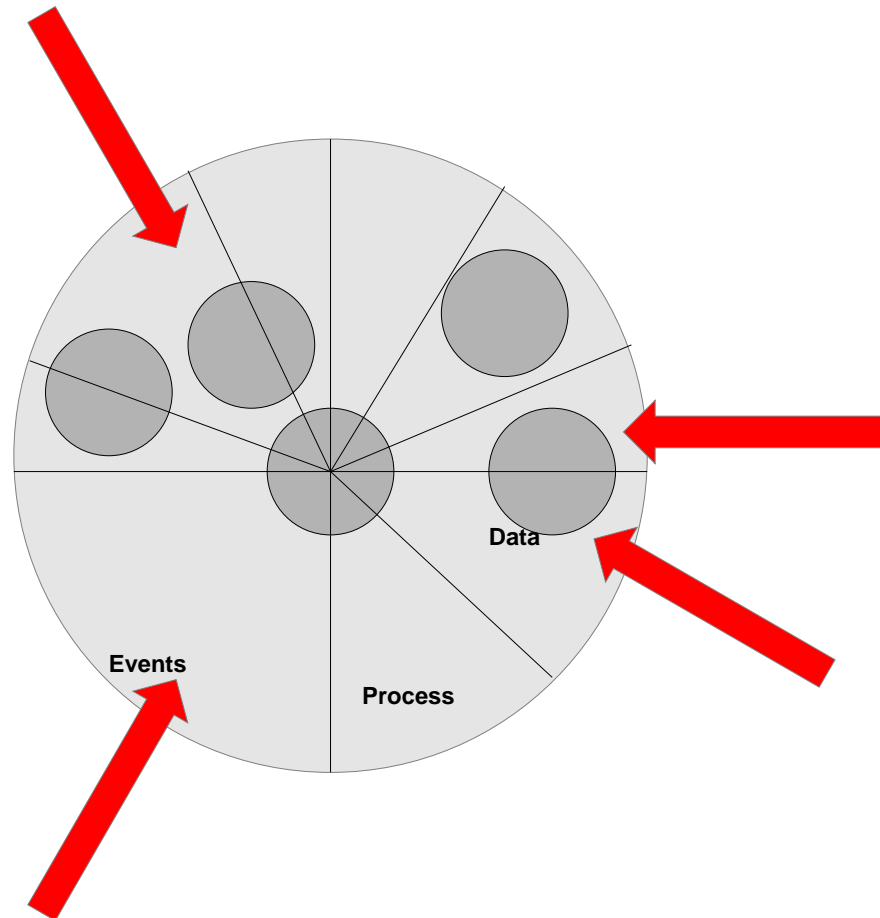
```
state = Initial;
for (;;) {
    e = getevent();
    switch(state) {
    case Initial:
        switch(e) {
        case Event1:
            ...
            state = New;
            break;
        case: ...
        }
        break;
    case ...
    }
}
```

'Spagetti!'

Processes Sharing Memory



One process per serial activity



Typical Server



```
handler(Req *req) {  
    ...  
    lock(l);  
    ...  
    unlock(l);  
    sendreply(rep);  
    threadexit();  
}  
main() {  
    for(;;) {  
        req = recvrequest();  
        threadcreate(handler, req);  
    }  
}
```

Processes in Shared Memory



By maintaining a pool of idle processes the number of process creations/deletions can be kept small.

Locking is hard to get right. Locks can be forgotten, they can be in the wrong place, they can lead to deadlock.

A better paradigm:

Coroutines



Coroutines are threads that *explicitly* relinquish control to other threads; only one thread runs at any one time.

If threads relinquish control only when shared data structures are in a consistent state, *locking is not necessary*.

But, when a thread is blocked on a system call, what about the other threads?

Solution One: Non-blocking system calls



Later versions of Unix, including FreeBSD and Linux, have non-blocking versions of many system calls. Calls like `read()`, `write()`, `listen()`, and `connect()` can be made to return immediately, allowing a thread to continue execution while the system call is blocked.

Results are picked up later with the `select()` system call.

`select()` is called with a list of file descriptors and a timeout and reports the file descriptor on which a system call finished along with the result of the call.

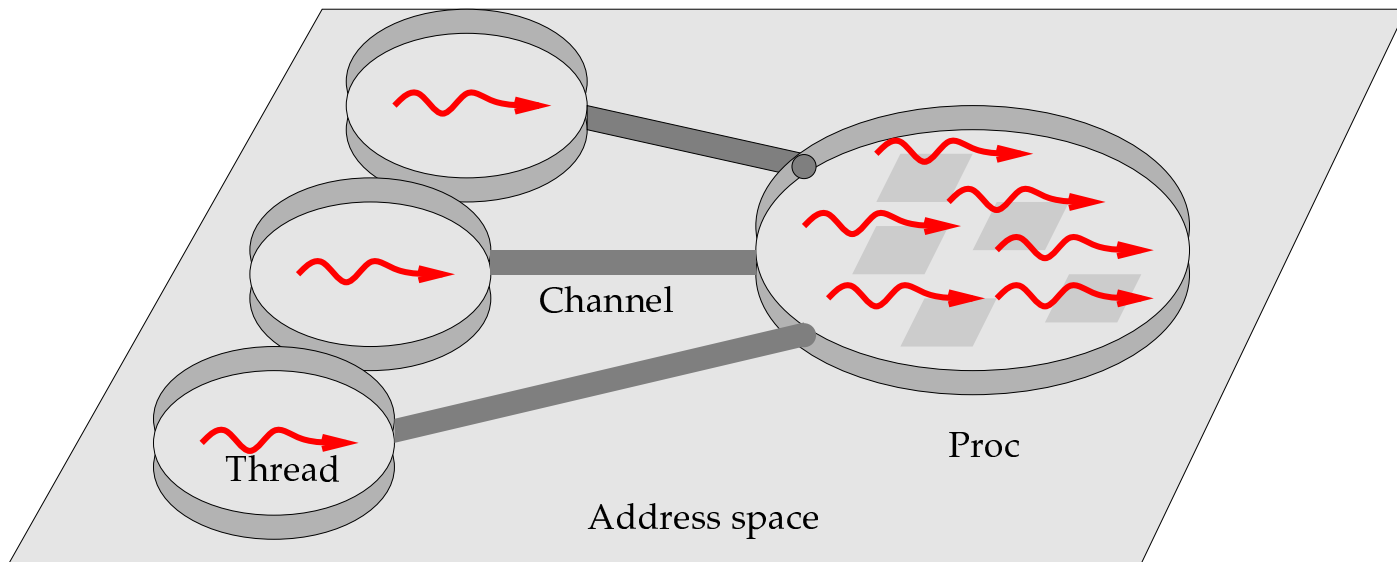
Solution Two: Procs, Threads and Channels



A *channel* is a message-passing mechanism: one process sends a value on a channel and another receives it. The first process to send or receive blocks until another process makes a matching call. They are implemented in shared memory.

Processes are scheduled independently (and may thus run concurrently on a multiprocessor) and share memory. Each process contains one or more *threads* which behave like coroutines (i.e., they are scheduled by the process).

Using Procs, Threads and Channels



Implementation in Plan9 (and Linux)



The Plan9 Thread Library makes the concurrent-programming model of CSP, Newsqueak, Alef and Limbo available to C programs.

The library implements *procs*, *threads*, *channels* and *locks*.

Channels



```
Channel* chancreate(int elemsize, int elemcount);
```

Chancreate creates a buffered or unbuffered channel. Unlike *newsqueak*, elements have no type, but each element in a particular channel must have the same size.

It is unbuffered or synchronous when *elemcount* is zero. If *elemcount* is non-zero, it specifies the number of elements that can be sent to the channel before it blocks.

Sending and receiving



```
int send(Channel *c, void *v);  
int recv(Channel *c, void *v);
```

Send writes the element pointed to by *v* to the channel pointed to by *c*. *Send* blocks until there is room in the channel to store the element (buffered channel) or until there is a receiver for the element (unbuffered).

Symmetrically, *recv* reads an element from the channel pointed to by *c* into the space pointed to by *v*. *Recv* blocks until there is an element in the channel (buffered) or until there is a sender (unbuffered).



```
typedef struct Alt {
    Channel *c; /* channel */
    void *v; /* pointer to value */
    Altop op; /* operation */
    Channel **;
    ulong;
} Alt;

int alt(Alt alts[]);
```

Alt sends or receives on at most one of a set of *send* and *recv* operations.



```
int f(Channel *c0, Channel *c1) {
    int v0; char v1;
    Alt a[] = {
        /*      c      v      op */
        {c0,    &v0,  CHANRCV},
        {c1,    &v1,  CHANRCV},
        {nil,   nil,  CHANEND},
    };
    for (;;)
        switch(alt(a)) {
        case 0:
            fprintf(2, "got int %d\n", v0);
            break;
        case 1:
            fprintf(2, "got char %c\n", v1);
            break;
        default: error("impossible");
        }
}
```

Typical I/O proc



```
void
mouseproc(void *mc) {
    char m[48];
    int mfd;
    Channel* mousechan = mc;

    if ((mfd = open("/dev/mouse", OREAD)) < 0)
        fatal("open /dev/mouse: %r");
    for (;;) {
        if (read(mfd, &m, sizeof(m)) != sizeof(m))
            fatal("mouse: %r");
        send(mousechan, &m);
    }
}
```

Typical Main Loop



```
a[0].c = chancreate(sizeof(m), 0);
proccreate(mouseproc, (void *) (a[0].c), STACKSIZE);

a[1].c = chancreate(sizeof(t), 0);
proccreate(clockproc, (void *) (a[1].c), STACKSIZE);

for (;;) {
    switch(alt(a)) {
    case 0: /*mouse event */
        threadprint(2, "click "); break;
    case 1: /* clock event */
        threadprint(2, "tic "); break;
    default:
        fatal("impossible");
    }
}
```

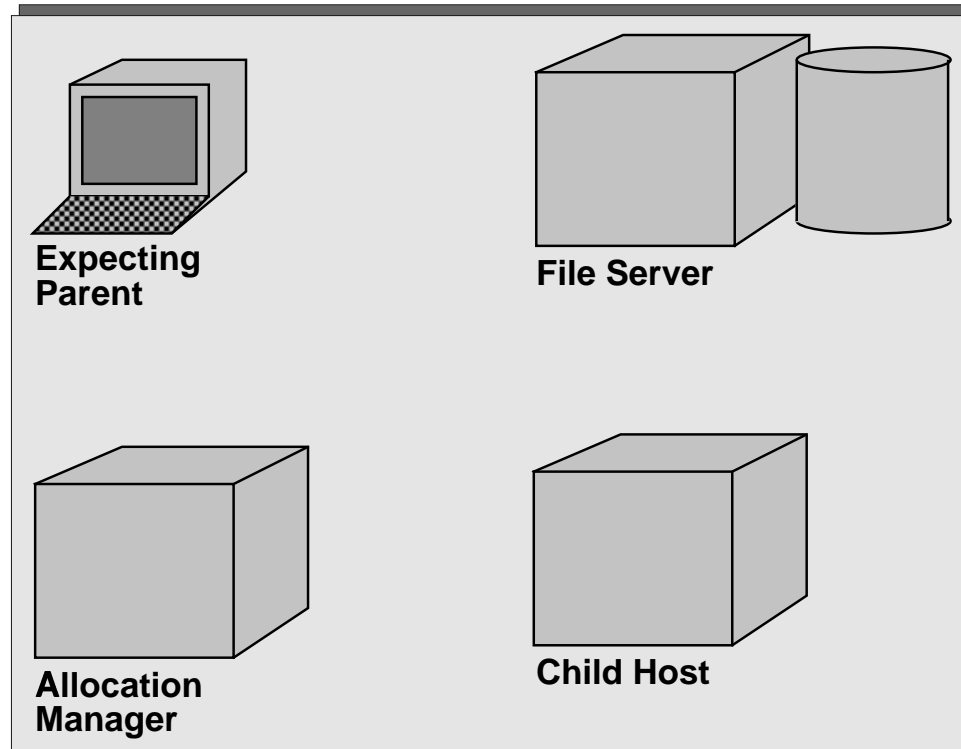
Consistency



If threads leave data structures in a consistent state before yielding, locks are not necessary and deadlocks become less likely.

Memory management is crucial. Be consistent about where you place responsibility for *freeing malloc*-ed data structures. Have a plan for cleaning up shared data structures.

Process Management in Distributed Systems



Interfaces for Process Management



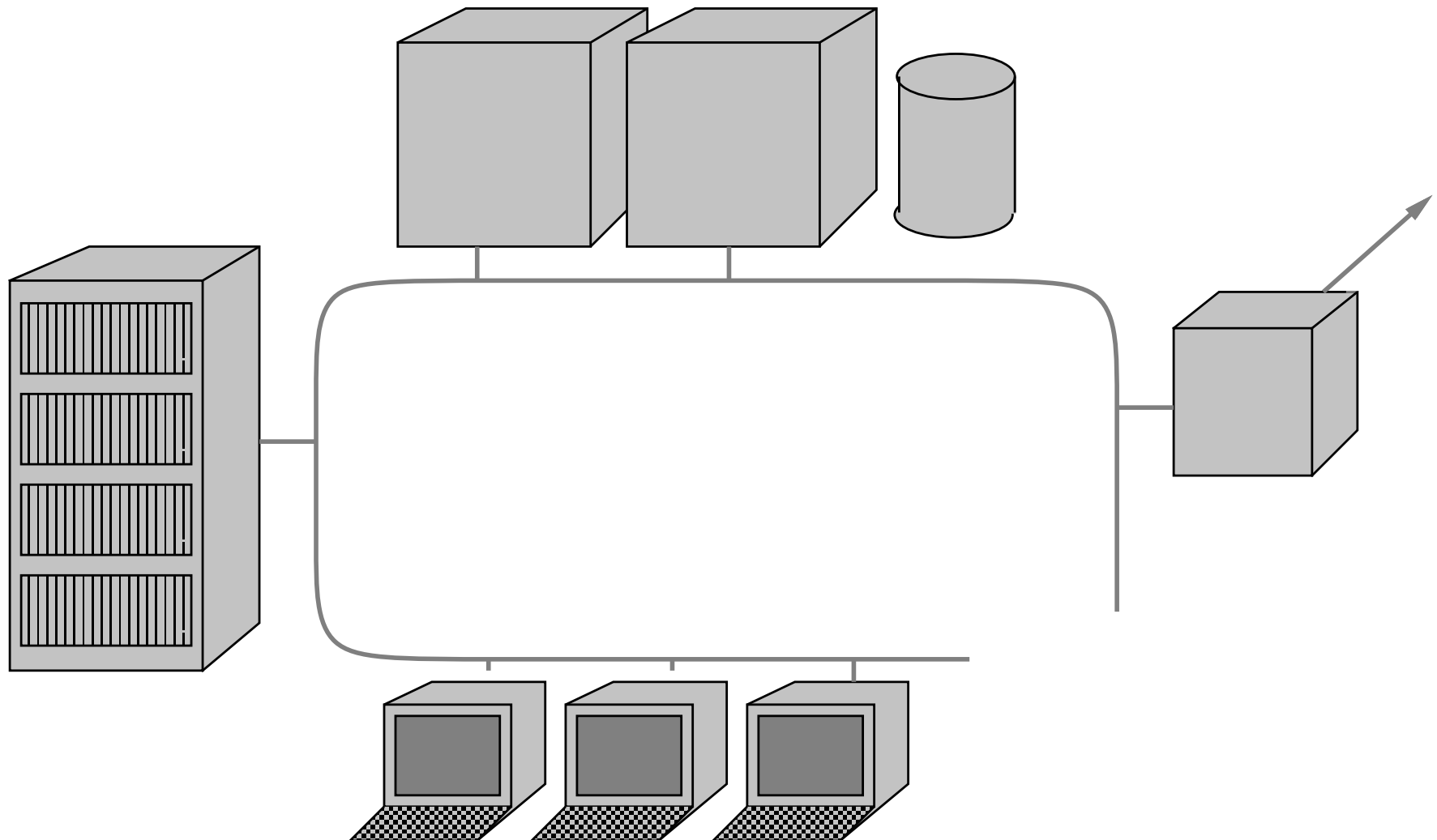
- *Operating-system interface* — process to system
- *Process-control interface* — parent managing child
- *Process-management interface* — system managing process

Process Management Functions

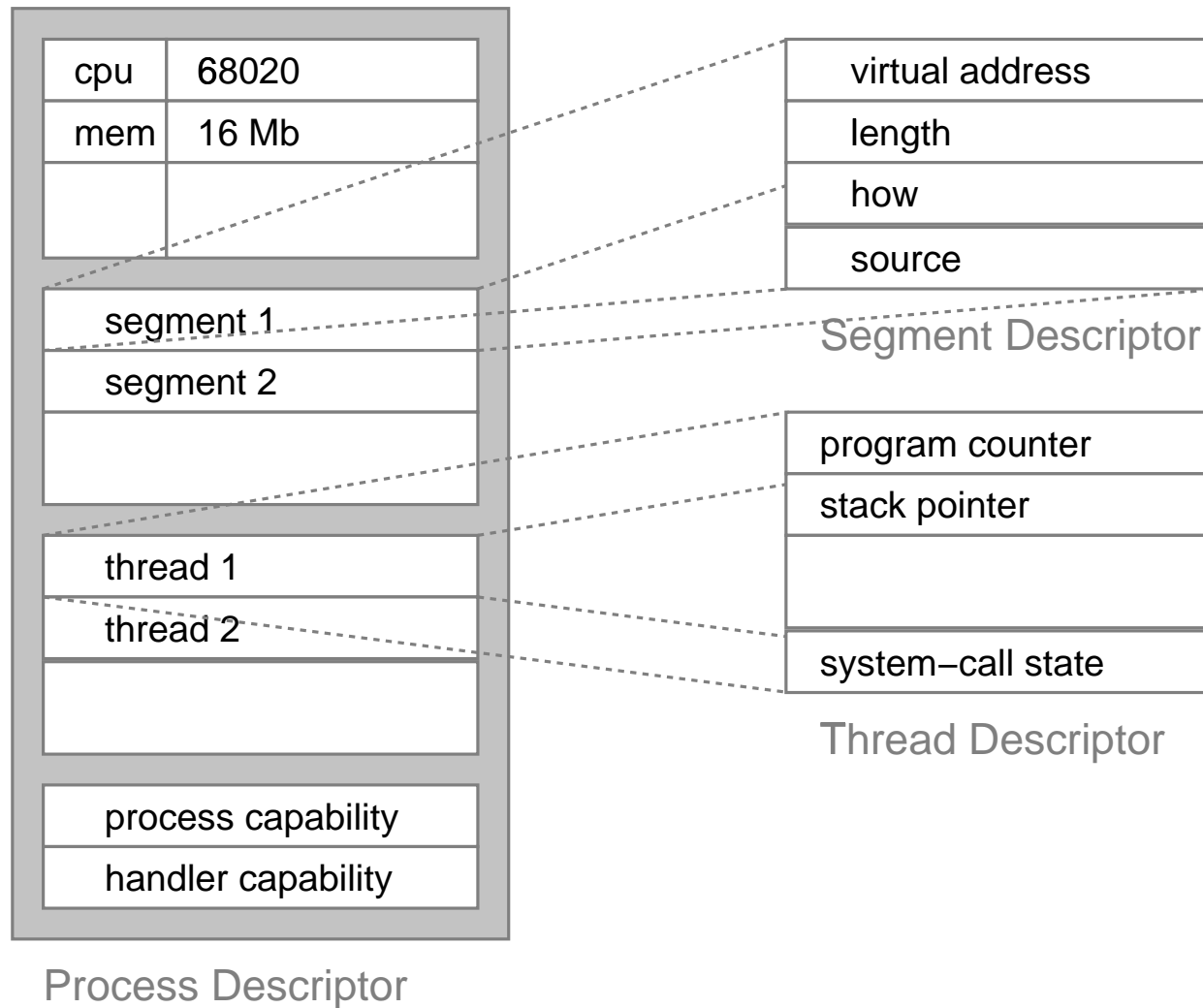


- Creation
- Destruction
- Debugging
- Checkpointing
- Migration

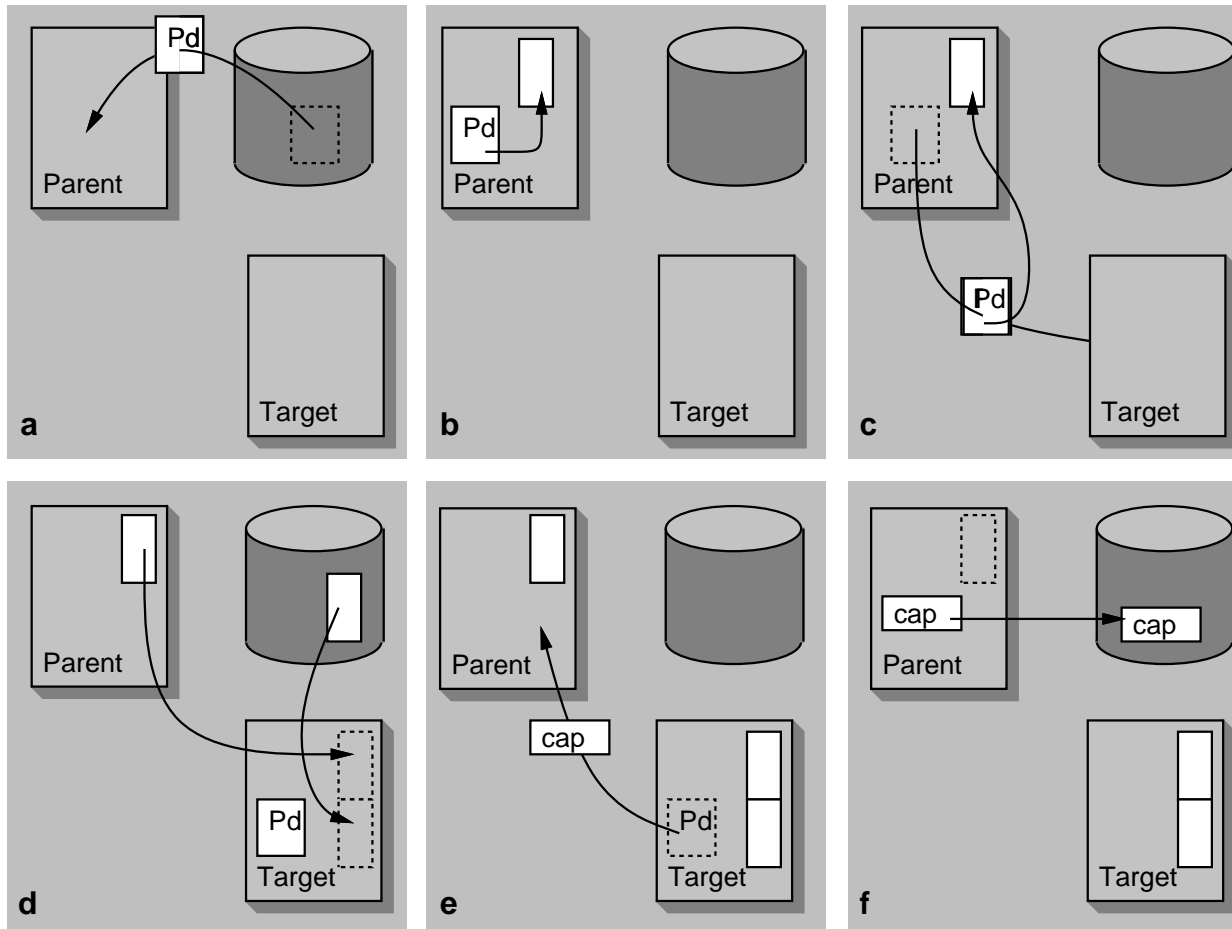
Process Management in Amoeba



Process Descriptor



Process Startup in Six Easy Steps





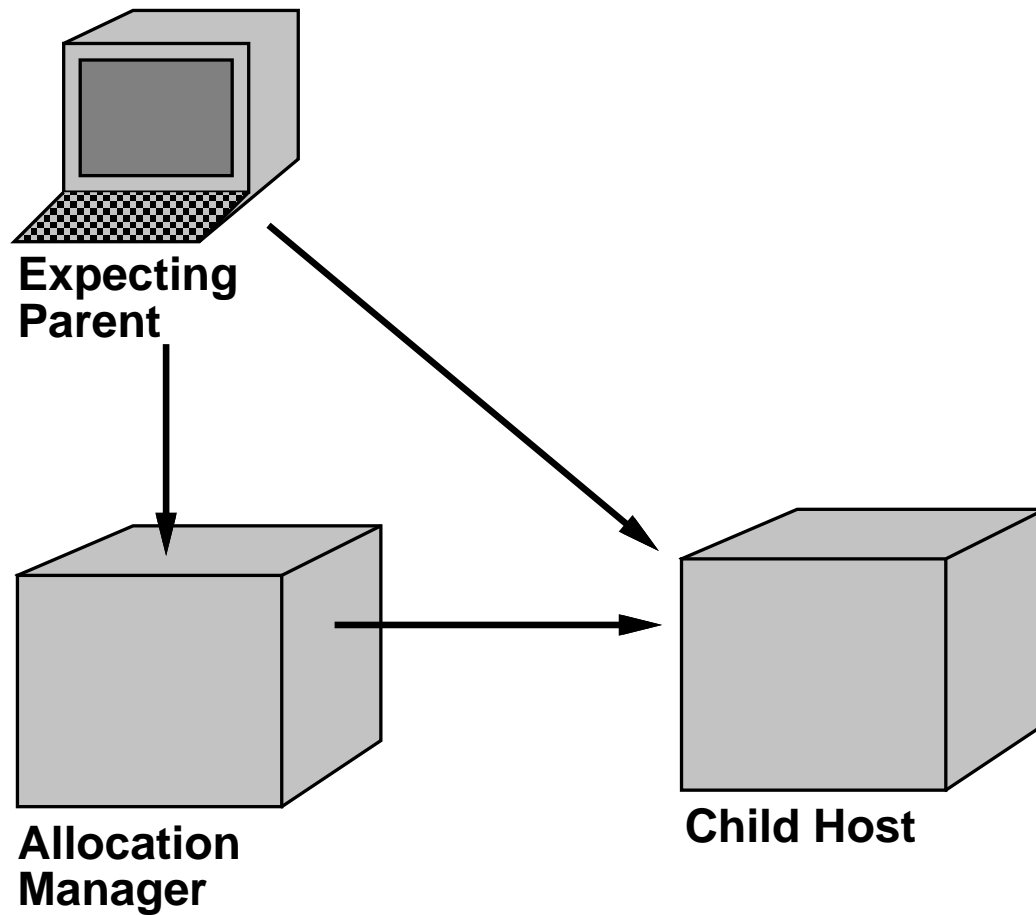
When a process is stunned,

- a process descriptor is made,
- incoming messages are discarded, but
- *try-again-later* messages are returned
- the process can be resumed or destroyed.

Management Interfaces



Process Control and Management Interfaces are Identical



Load Balancing



Balance load on hosts in the network

Methods:

- Allocate at process creation
- Re-allocate dynamically (migration)

Availability of enough physical memory often more decisive than availability of CPU cycles.

Load Balancing and Migration



Migration takes time — a process' memory contents need to be copied.

On Ethernet, a 5 Mb process takes at least 5 seconds to copy.

Most processes do not live as long as 5 seconds.

Is Migration Useful?



Only marginally for load balancing

But useful when idle workstations are used as a processing resource

Also useful when hosts have to be taken down

Speeding Up Migration



Reduce memory copy times by executing while copying

CMU method: migrate process descriptor and page-in over the network

Stanford method: Migrate memory and use “*dirty bits*” to detect modification by still running process. After one or two sweeps of dirty pages, migrate process descriptor and copy remaining pages.

CMU method is simpler, but leaves pages all over the place. Only the working set is copied, though.

Stanford method gets more parallelism of processing and copying, but spends even more time copying pages than straightforward migration.